Hardware/Software Co-design of Schedulers for Real Time and Embedded Systems

By Jorge Ortiz

Engineering of Computer Based Systems

Electrical Engineering and Computer Science Masters Program

UNIVERSITY OF KANSAS

May 2004

Abstract

Embedded systems can no longer depend on independent hardware or software solutions to real time problems due to cost, efficiency, flexibility, upgradeability, and development time. System designers are now turning to hardware/software co-design approaches that offer real time capabilities while maintaining flexibility to support increasing complex systems. Although long desired, reconfigurable technologies and supporting design tools are finally reaching a level of maturity that are allowing system designers to perform hardware/software co-design of operating system core functionality such as time management and task scheduling that allow the advantages of higher level program development while achieving the performance potentials offered by execution of these functions in parallel hardware circuits.

This thesis presents the hardware/software co-design, implementation, and testing of the event scheduler and timer services provided by the KURT-Linux real time operating system in a Field Programmable Gate Array (FPGA). All event-scheduling functionality was migrated into hardware with a standard FPGA-based address mapped register set interface for the remainder of the operating system. This hardware-based event scheduling functionality liberated the CPU from performing the overhead processing associated with managing event queues, and provided microsecond resolution scheduling of events.

The scheduler was implemented and tested in different architectures to show the portability and reliability of the co-design solution. Worst-case scenarios for expired event execution times were bounded with the use of hardware-enabled scheduling, and performance tests for the hardware/software solution yielded the same results as the software implementation. This work represented a critical first step towards achieving a full hardware/software co-design of key operating system functions into a hybrid system for embedded applications.

Acknowledgements

I would like to thank my advisor and committee chair, Dr. David Andrews, who initiated me into research and specifically introduced me to this project. His guidance and supervision helped me navigate through graduate school, and his encouragement directed me towards higher levels of education.

I am grateful to my other thesis committee members, Dr. Perry Alexander, for helping me decide on an area of focus for my education, and Dr. Douglas Niehaus, for initiating much of the research projects I've been involved on.

I wish to also thank my project teammates Sweatha Rao, Mitch Trope, and Mike Finley for their contributions, support and feedback for the design and implementation of this project.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

1.1 Problem domain

One of the constant challenges for real-time system designers is building a platform that can meet the timeliness requirements of the system. These requirements include time deadlines for scheduling tasks that cannot be missed. Additionally, the scheduling resolution of these deadlines are many times at a finer granularity than what commercially available software-based schedulers are able to provide.

Many commercially available operating systems schedule events based on a periodic interrupt from a timer chip, known as the heartbeat of the system. Due to the unique timeliness requirements of real-time events, this heartbeat approach is insufficient in both frequency and resolution. Basing real time scheduling decisions on a periodic scheduling approach can yield unacceptable performance due to the overhead processing associated with context switching times and aperiodic interrupt processing requirements. In fact, under certain conditions this heartbeat approach can introduce so much overhead, that the system not be able to achieve any useful computation [01].

Another problem results from the current approaches in use for reducing the service time of an interrupt service routine once acknowledged. In fact, there has been little research done on measuring and reducing the delay between the time an event is set to be scheduled and the time at which the event actually is executed [02]. This is not a trivial problem as this delay is difficult to measure using software based monitoring approaches. To accurately capture this delay, sophisticated and expensive logic analyzers are required. Even with a sophisticated and expensive analyzer, it is still time consuming and difficult to capture this delay time [02].

Finally, real-time systems must track and maintain timing information for multiple events that should be serviced in the future. Typically this is done in software with the events stored in a priority queue. Maintaining and sorting the queue introduces time delays and jitter but is still necessary to ensure that the highest priority event will be serviced before its deadline. Sorting through the event priority queues is time-consuming and variable. Uni-processor solutions introduce overhead to manage the queue structure. Multi-processor solutions have the additional problem of communication overhead that can add even more unpredictability to the system.

1.2 Thesis motivation

This thesis introduces a new hardware/software co-design approach for real-time systems that require fine-grained scheduling support that may not be achievable using software-only solutions. Our approach exploits the performance advantages of hardware/software co-design that integrates parallelization of independent functions in dedicated and custom hardware working in conjunction with software running on a processor.

The policies presented in this thesis were developed and tested in different Field Programmable Gate Arrays (FPGA's), and can also be realized in Application-Specific Integrated Circuits (ASIC's). Further, our work can be extended for fine-grained performance monitoring. Our approach realizes the scheduling queue and operations on the event queue in hardware. This approach minimizes the overhead time associated with adding elements to the queue, a characteristic useful when logging events in a system while minimizing the instrumentation effect of the measurements. One such example is Data Streams and their logging mechanisms, DSKI [03].

1.3 Contributions of this thesis

The work presented on this thesis is part of the KU RTFPGA (University of Kansas Real-Time Field Programmable Gate Array) project. The goal of the KU RTFPGA project was to migrate key operating system functionality into hardware. The initial work of Mitchell Trope and Sweatha Rao migrated time keeping into the FPGA. The contributions of this thesis are:

1) The design and implementation of the software/hardware interface between the KURT library and the FPGA to store event information in the FPGA's Block Ram memory,

2) The design and implementation of a memory manager entity that generated addresses for queued events,
3) The design and implementation of supporting structures for a hardware-based priority queue implementation, and
4) The implementation of the scheduler for the events stored in this hardware queue.

Sweatha Rao and I collaborated to implement additional event queue functionality for searching the event queue to find the earliest deadline event, and delete events from the queue. We also collaborated on the design and implementation of additional debug and control functionality.

2. Background

2.1 Prior work in hybrid hardware/software co-design

Improvements in processor clock speeds and memory size have provided continual incremental performance enhancements to existing desktop applications. Additional incremental enhancements have resulted from new architectural techniques associated with out-of-order execution and instruction-level-parallelism, and deeper caching. Both technology and architectural advancements have resulted in increased throughput of multiple time sliced programs: the design objective of desktop systems. These techniques however, can have adverse effects on the worst-case execution times of individual programs. While not an issue for desktop systems with no time deadline constraints, guaranteed worse case execution time, or turnaround time, is of vital importance for real time and embedded systems. The idea of having to go down through multiple cache levels into a physical memory due to cache misses, or executing code out of order to achieve a higher aggregate combined throughput can result in missing scheduling deadlines and be catastrophic. Counter to the desktop architectures developed for general-purpose systems, different platform architectures are needed for real time systems that focus on minimizing the worst-case execution times to support real-time constraints [04].

2.1.1 Parallel systems

Parallel systems have emerged in response to meeting tight timeliness requirements. These include multi-processor systems, application-specific integrated circuits (ASIC), field programmable gate arrays (FPGA), systems with several chips on a die, system-on-a-chip (SoC), and dynamically reconfigurable systems. The commonality within all of these systems is to exploit the parallel capabilities of the added hardware functionality. ASIC's provide application specific specialized hardware, while FPGA's provide a programmable sea of gates that can be configured and modified for multiple purposes. Systems with parallel subsystems on a die can provide advantage of the processing power of other processors running in parallel, while a SoC usually has a single processor, which takes advantage of the FPGA's flexibility by an interconnection between the processor and the FPGA. Finally, the dynamically reconfigurable systems can set themselves in different configurations for a particular situation.

2.1.2 Hybrid systems

The classic approaches for hardware acceleration generally fall into the following three non-disjoint categories: 1) Exploiting recursive patterns, 2) increasing quality of service, and 3) meeting real-time constraints [05]. It is intuitive how parallel hardware can trade space for time to exploit independent recursive and iterative software loops. Iterative and parallel loops that repeat code segments on independent data sets require valuable clock cycles to implement the same operations on the different data sets. As an example DES encryption streams data through complicated bit shift and XOR operations with a key, 16 rounds per piece of data. By unrolling and pipelining the operations into parallel hardware, it is possible to perform a simple space-time tradeoff that results in a linear reduction of the encryption time [06]. Improving the quality of service is also evident from the previous example. A well-designed hardware-based encryption system should be able to encrypt/decrypt different 64-bitdata every singe clock cycle, hence increasing the throughput of the system. An example of hardware being used for quality of service can be seen at the backend of network systems where the computation of intensive scheduling decision logic is often needed for controlling data movement in the form of packets [07][08]. Finally, hardware has been used as an accelerator for time critical real time functions. In this scenario, computations are executed in parallel and controlled by the system software that ensures all timing constraints are met. Finally, projects such as HERT at the University of Kansas are exploring migrating low level system processing, such as interrupt service routines and device handlers that introduce non-deterministic overhead, in hardware.

2.1.3 Real time operating systems

Over the last decade, operating systems for real time platforms have become available that offer the advantages of platform independent and system programming using higher level languages. It is now common to find commercially available real time operating systems for many hybrid systems. Real time operating systems (RTOS) such as Wind River's VxWorks provide generic application program interfaces (APIs) for file system support, I/O Management, and scheduling. A more compact version of a RTOS is a micro-kernel, which provides runtime applications with real time services through its interface with the system resources [09]. Unfortunately, these solutions mostly are targeted towards ASIC's which need specific resources or have special limitations. A more viable and familiar solution for programmers would be to provide Linux with real time support. Several projects exist for extending Linux into the real time domain. Examples include Linux RT [10], RT Linux Toolkit [11], and KURT-Linux [02]. The first two achieve real time behavior by providing a real time execution that treats Linux as the lowest priority task in the system. KURT Linux provides real time packages to the Linux source tree, thereby achieving real time capability from within Linux. The latter is our RTOS of choice, and the one that we chose to perform our hardware/software co-design.

It was our goal to enable/facilitate key OS functionality within parallel hardware circuits [12]. Specifically our goal was to migrate time-critical software code from the kernel into the hardware that would allow a system using KURT-Linux to meet the stringent of hard time constraints. Our initial targets were time critical functions associated with interrupt handling, task scheduling, memory management, resource allocation, and data routing.

2.2 FPGA hardware/software co-design

The ability to fabricate smaller and smaller transistors has had significant effect on computer system design trends. Moore's law, which states that the processing speed of a CPU doubles every three years has continued to hold true. Moore's law is also being followed by programmable devices. Figure 2.1 below [13] shows the change on design trends over two decades of embedded systems evolution.

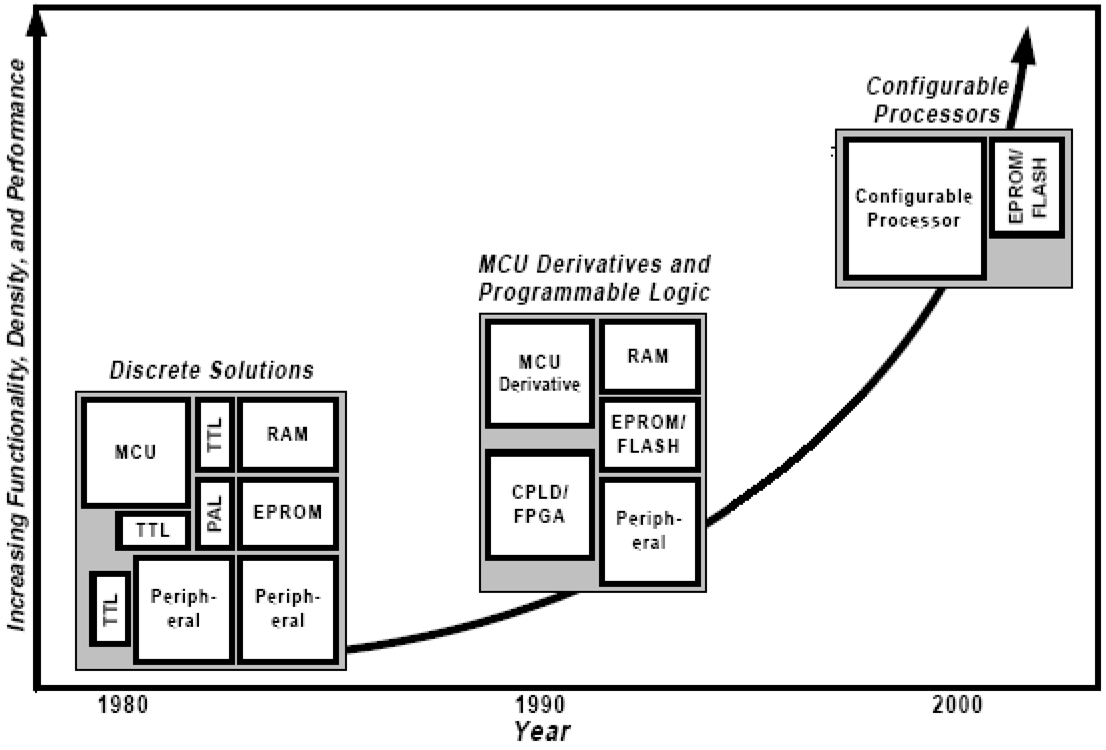


Figure 2.1: New programmable technologies of increasing density and performance

In the early 1980's embedded systems were built using discrete components. The Micro Controller Unit (MCU) was the central processing unit and Transistor-

Transistor- Logic (TTL) devices or Programmable Logic Devices (PLD’s) provided the interface between the controller and its discrete peripherals [14].

By the late 1980’s and start of the mid 1990’s, companies started using MCU derivatives like Complex Programmable Logic Devices (CPLD’s) and FPGA’s, which provided control and some functionality for MCU peripherals. The MCU was integrated with additional system RAM and boot PROM to create integrated versions of the earlier MCU’s.

With the development of Intellectual Property cores now provided by companies such as Xilinx and Altera, and the increased capabilities of FPGA and CPLD devices, entire systems are now being built on a single silicon die (SoC's). These SoC’s, which can be customized or configured for specific applications, reduce the economic disadvantage and inflexibility associated with ASIC customized designs, but still provide customization. [14]. The most current offerings for SoC's such as the Virtex II Pro [15] and Excalibur [16] now provide a dedicated processor and programmable logic on a single configurable chip [14]. Commercially available IP cores such as UARTs and device controllers can be incorporated and even tailored into an existing SoC chip within the FPGA fabric. These platforms represent a robust environment for development of wide ranging and changing application requirements.

Design Goals for individual components

FPGA's provide flexibility in custom hardware circuit design. With these new hybrid chips, it now becomes viable to execute parallel processes running in both the software and in the FPGA hardware. In fact, an arbitrary number of parallel processes could be running in the FPGA along with concurrent processes on the CPU. Researchers are now exploring approaches that support the dynamic migration of parallel and concurrent processes fluidly across the CPU and FPGA components [17].

2.2.1 Desired embedded control system properties

Real time computers control larger embedded systems. Because of the close interaction with physical components and communication between systems, several properties are required for a real-time embedded control system. These are timeliness, concurrency, liveness, interfaces, heterogeneity, reliability, reactivity, and safety [18] [05] [19]. Timeliness is the property that is concerned with total executions times. Therefore timeliness considerations must account for not only the time for execution of instructions, but also all other system delays times such as communication times. This is essential if the system has concurrent processes running in parallel that must synchronize or react to asynchronous stimulus between processes. Liveness is the property that the system must never stop running, either by halting, suspending or terminating. This scenario would be deemed defective if implemented in hardware [19]. Component interfaces must be well defined, not just for their static unification of component communication ports, but also for the dynamics of computations occurring between them and their timing [20]. The heterogeneity of the system comes into play during the dynamic part of inter-component communication, as different components need to "talk" to each other in an understandable way. Hence, either software must accommodate to receive a constant stream of computation from a hardware process, or hardware must expect discrete results from a software procedure, or a tradeoff between the two must be achieved. The communication between parts of the system, and the level of correct system behavior, must be reliable. This is because the system will have high reactivity to the events happening around it in real-time. Finally, safety is always an issue in embedded and real time systems, since the system is probably in control of expensive or invaluable objects, and a failure could even result in a great loss of life.

2.2.2 Design considerations for HW/SW co-design

Solutions for problems with time consuming simple and constant repetitive processes, like network packet routing, data encryption, mathematical matrix operations, multimedia encoding and high speed signal processing are very suitable for hardware based implementations. Problems involving slower, more complex, variable computations, like software applications, GUI displays and end-user input processing are more appropriate for software [04]. Hence we observe that while different goals might require different parts, these are clearly not disjoint, and an effective system design would use both hardware and software to achieve a combined purpose complying with system design restrictions.

2.2.3 Average case / worst case

Real time systems have specific time constraints that must be considered during the initial design process. The average case delay of the system becomes of secondary importance against the worst-case scenario. Care must be taken in finding worst-case execution time paths in a real-time system to ensure its correctness. All variability in code execution and system processing times must be identified and accounted for to determine the worst-case execution time. The worst-case execution time is more easily achieved if the system has no influence from outside devices. However, a primary feature of most systems nowadays is the ability to communicate to peripherals running in different time scales, receiving and handling signals from sensors, controlling outside actuators, and receiving interrupts to conduct more critical operations, etc. These functions can have very broad minimum and a maximum processing times. Not only is quantifying these time gaps important in determining worst-case time, but even if known can still degrade the overall utilization factor of the system. [21].

2.2.4 System predictability

If the system is running a complex operating system, the number of states it can reach can be significant. However, if we concentrate on the hardware components, a certain degree of model state checking can be achieved. This is especially true for ASIC's, but FPGA's with a companion chip on a SoC should also be predictable. This checking can be guaranteed under normal conditions such as receiving events from the outside, be those an ASIC receiving a signal for a pushed button on a vending machine, or a CPU memory request to a hardware peripheral inside an FPGA. In this situation, model checking of the system can be achieved through formal verification analysis tools like Spin [22] or Rapide [23], using process description languages like Promela [24].

2.2.5 System flexibility and performance

Another design consideration for a hybrid system is flexibility. FPGA's are configured at synthesis time, while dynamically reconfigurable hardware [25] will eventually allow reconfiguration at run time. This flexibility can be used to adjust the design for die area, system speed, resource allocation, power consumption, and system performance. While designers want their constraints to be met and fulfilled, they also want the system to achieve useful computations, hence why a certain level of system performance is also required. In fact, designers will want the same or better level of performance of a non real-time operating system [26].

2.2.6 Co-design settings for programmers

One of the drawbacks for system co-design is the lack of support for software programmers trying to use embedded technology. Several projects have been created to ease the transition into the real time world. Handel-C is a high-level language based on ISO/ANSI-C for the implementation of algorithms in hardware [27], SystemC provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++ [28], and the Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems, focusing on embedded systems mixing hardware and software technologies [29]. These ongoing projects provide different environments to develop hardware enabled/optimized pieces of code for their targets [30].

2.3 Scheduling

As specified in the section 2.1.3, attractive uses for hybrid software/hardware real-time operating systems include interrupt handling, task scheduling, memory management, resource allocation, and data routing. Of particular interest to us are the cases for interrupt handling and task scheduling.

2.3.1 Scheduler operation

When we have multiple levels of processes working “simultaneously” on the same system, we need to schedule certain shared resources (CPU, memory) in a fair manner. A scheduler takes care of managing the processes/entities, the resources they are requesting, and running a scheduling decision/algorithm to assign resources to requesters for a specified/unspecified amount of time. However, such a decision does not come for free, and oftentimes, even efficient resource allocation scheduling algorithms end up performing worse than a random scheduler. This can result from the decision function itself, as it requires resources from the system in order to run.

The scheduler functionality is usually dependent on *how* it's allocating *which* resources *to whom* and for *how long*. The most common task for a scheduler is allocating CPU time to different programs, threads or events, for an organized amount of time to ensure fairness and freedom from starvation, guaranteeing every program will get a CPU share [31]. This allows for uni-processor multitasking models that allow different programs to run at different times in a form that simulates concurrency. As the number of processors a scheduler must manage increases, so does its internal logic and communication costs.

A scheduler for real-time systems must handle real-time events, including stimulus from the outside world collected through sensors, or a high priority event. In all cases, these events must be given special importance, and the execution of the event be met within its time constraint. The adopted method to handle these situations is to create an interrupt for the CPU, and run the interrupt service routine. This is a costly process, particularly if a system has to support multiple real-time interrupts or handle fine granularity events scheduled to run within a small period of time. Handling these events has the unfortunate cost of CPU cycles, the very resource that is in demand. As previously indicated, if the scheduling decision is complex, the larger the cost will be during the scheduler runtime. Such an overhead to an already overloaded real-time system is not desirable.

2.3.2 Scheduler components

A scheduler in general has five main components, which vary slightly from implementation to implementation. The first component of the scheduler defines what the event type is: an interrupt, a program thread, an object, etc. However, we only need to know what the event is, not what the event does. Handling the event is typically left for a different software component within the system. The second component is the queue structure for the events. This component takes care of storing event information for each event and allowing events to be added or removed. The next component is the scheduling algorithm, which is the decision function to choose the next event to be scheduled. The most common algorithms are Earliest Deadline First (EDF), Rate Monotonic, and Priority Based. The way in which the scheduling algorithm is implemented inside the event queue is considered the fourth non-trivial component. This is because sorting operations can take a large amount of computational time, which is in itself crucial to real-time systems, and lead to hardware supported sorting designs [32]. The final component of a scheduler is the interface back to whatever requested a scheduler service, in general a CPU. This can be achieved through interrupt controllers, message passing or shared memory.

2.3.3 Hardware scheduler research

A fair amount of research has been done in the area of migrating schedulers and time critical application programs into hardware for real time systems. The most straightforward approach is to implement hardware support for real time applications that cannot execute fast enough in software. Here, the scheduler will assign certain special priority real-time tasks to dedicated hardware in the system. The next approach is to implement the scheduler in hardware, through an FPGA. The scheduler is usually static, but some systems allow dynamic scheduling algorithms, which can be changed during run-time [33]. Allowing reconfiguration in the FPGA's also brought forth different scheduling techniques for non-static schedulers [34], and with the arrival of multiprocessor machines, new schedulers sprung forth to maximize resource utilization, with the main function of being hardware accelerators [21].

Most of these approaches have undesirable limitations in their implementation. One vital but often overlooked property of most systems is the way the processing of interrupts are handled. While most of these systems implement optimizations for handling interrupts once an interrupt is detected, they fail to notice that there is indeed a non-trivial amount of time in which the interrupt flag is set and waiting to be noticed by the CPU, which most likely won't happen until the next scheduling decision function iteration. However, providing a finer granularity for the scheduler means paying a higher overhead of computation on the system. The reconfigurable solutions can provide better performance at the cost of hardware reconfiguration overhead, which might not be possible in a real time system. And lastly, multiprocessor solutions that handle scheduling are not ideal, since there would be certain unreliability in the system due to communication and synchronization costs. These tradeoffs in the designs are limiting factors for possible deployment of real-time systems.

The KU RTFPGA project concentrates on providing fine grained scheduling resolution and minimizing the overhead payment with FPGA backend support [35]. This is accomplished by using the fine resolution real-time operating system KURT [02], and implementing the interrupt scheduling in an FPGA. Further developments in this project include moving the program/thread scheduler into an FPGA [36].

3. RTFPGA: Project overview

The objective of the KU RTFPGA project is to minimize the overhead processing introduced by the operating system by migrating key functions into hardware. The first function that was identified was time keeping. This was achieved by migrating the jiffy and sub-jiffy registers into the FPGA. After the timers were migrated to hardware, the scheduler was identified as the next function.

3.1 – Design approach

Our design approach was to migrate a shadow event queue and associated processing for entering, sorting and deleting entries. As our first step, we did not migrate the scheduling algorithm into the hardware, but instead targeted the bookkeeping operations performed by the scheduler. Once the event enqueue and dequeue operations were accomplished, a more sophisticated way of accessing data within the queue was needed for the delete and sort operations.

Due to the linear nature of the event data stored in hardware, a circular scheme was used to loop through event entries in the queue linearly, comparing them to the value being deleted. Once this was in place, we used the same scheme to continuously retrieve event information and run it through our scheduling decision function. The decision function then forwarded its scheduled event towards the timekeeping registers, and upon a timing value match between the two, an interrupt signal was generated to the CPU indicating an expired event.

By using this approach, the CPU is relieved from accessing, maintaining, sorting, searching or polling the queue to find the next scheduled event. This work can occur in parallel to an application running on the CPU, and consequently the CPU will only be interrupted when needed and without having to manage the event queue, like presented in Figure 3.1 below:

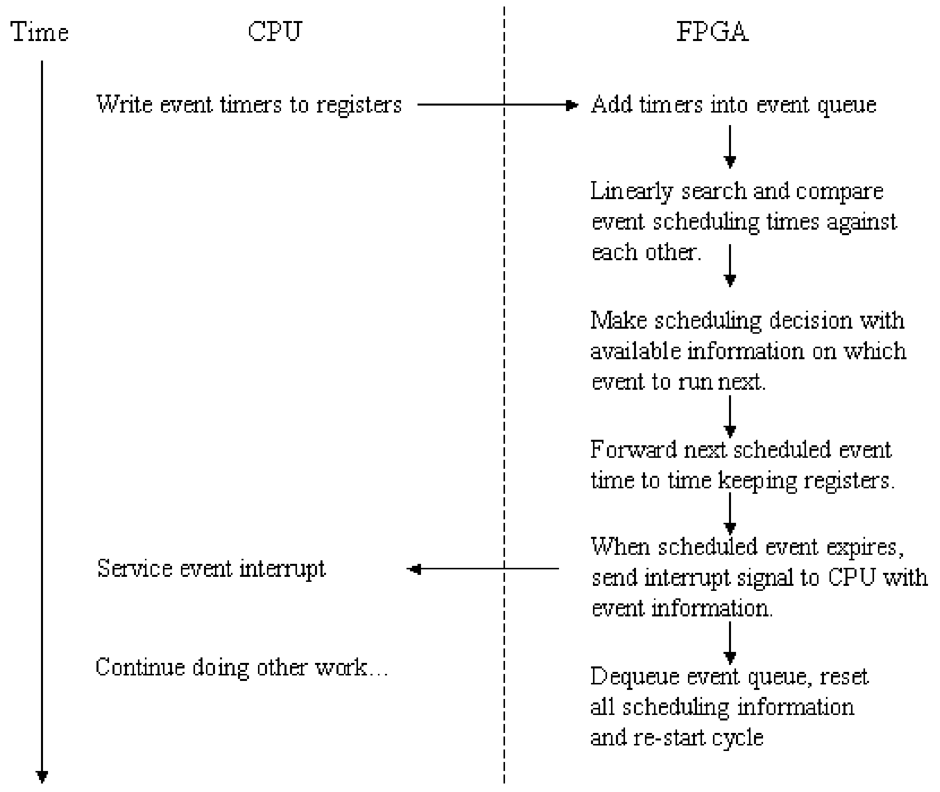

Time
CPU
FPGA
Write event timers to registers
Add timers into event queue
Linearly search and compare event scheduling times against each other.
Make scheduling decision with available information on which event to run next.
Forward next scheduled event time to time keeping registers.
When scheduled event expires, send interrupt signal to CPU with event information.
Service event interrupt
Continue doing other work...
Dequeue event queue, reset all scheduling information and re-start cycle

Figure 3.1: CPU/FPGA Event Scheduling Flowchart

3.2 – Design functionality

This event queue functionality was implemented in the FPGA by allowing the CPU to write the event information to an FPGA-mapped memory address. This information includes both the event scheduled time and a reference pointer to the function to call upon running the scheduled event. After the FPGA stores this information in a local event queue, it runs an algorithm to identify the next event to run, and sends its event time to a match register next to a free-counting hardware clock. Currently, the scheduling algorithm used is Earliest Deadline First, which is simple to implement without demanding additional logic. Upon a match between the scheduled event time and the actual board time, an interrupt is generated and sent to the CPU, for it to service the event by calling on its reference pointer.

Now that the event with the earliest deadline has been scheduled, it's popped out of the queue, the FPGA resets it's scheduling information, and a new event is found to run by retrieving timing information for the remaining events.

Further, due to different time scales, the CPU can not read interrupts as fast as the FPGA can generate them, so a second queue was implemented in hardware, a FIFO queue. The purpose for this is just to store the scheduled events until the CPU can read them.

Finally, we provide deletion operations in our queue, by specifying event information for the event to be deleted to the FPGA. This is useful functionality since several interrupt generating events come in the shape of watchdog timers. These are generally deleted before their deadline is met, or are replaced by other watchdog timers.

3.3 – Previously implemented FPGA modules

3.3.1 – Default Intellectual Property FPGA Blocks

Our original development platform was the ADI Engineering's 80200EVB board with a Xilinx Spartan-II chip. Our Xilinx Spartan-II board came with existing Intellectual Property (IP), the VHDL hardware design language source files, to perform memory management, bus arbitration and I/O interfacing. The memory manager performed data transfers and generated chip decode logic for the SDRAM chips as well as performed periodic refresh of the SDRAM cells. The existing IP was first modified to allow us to create memory-mapped registers inside the FPGA. This was implemented in a new hardware module, the `frp` (FPGA request processor) that effectively intercepted all memory operations on a specified, unused SDRAM address range. Secondly, we created a `Utime` module that contained a timer that incremented at the memory clock speed, 10 ns. Throughout the rest of this thesis, when we refer to a module, we refer to the hardware implementation for some functional block inside the FPGA.

A block diagram of the SDRAM memory request intercept hardware is shown below:

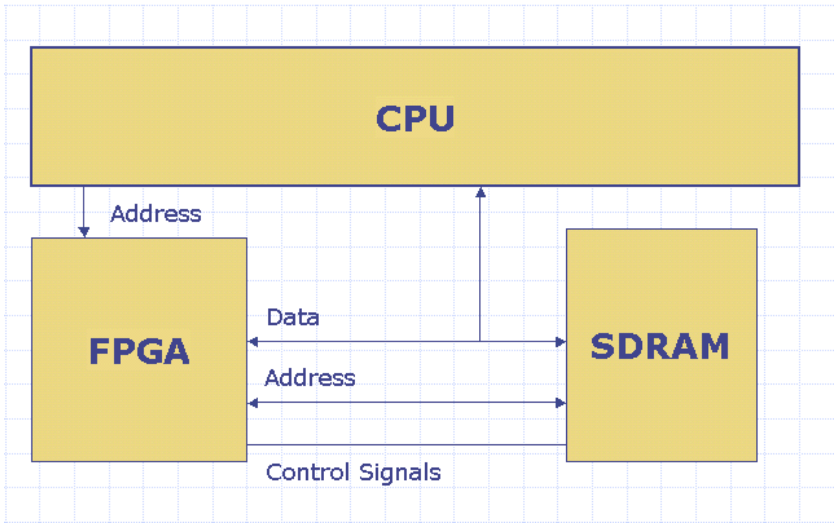


Figure 3.2: Hardware memory controller layout

The next presents an overview of the existing modules, explained individually afterwards.

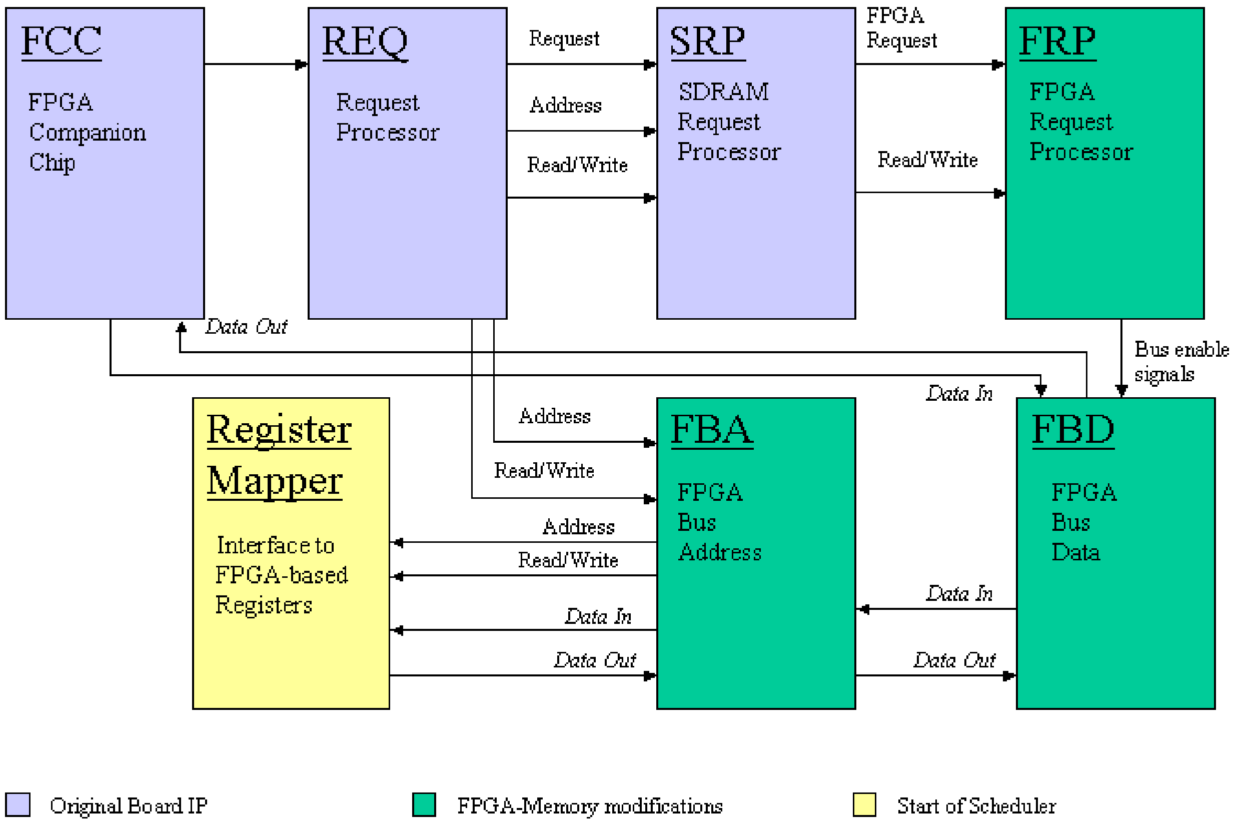


Figure 3.3: FPGA Memory *module layout*

SRP Module

The SDRAM request processor (SRP) filters SDRAM requests, and checks other address ranges for peripheral requests. Upon a peripheral device request, it forwards information on the data bus and generates the required signals to handshake and control the appropriate peripheral.

3.3.2 – FPGA memory request implemented modules

The FPGA implementation is comprised of four modules, each with specific tasks and interfaces to other modules. The previously implemented modules are the only architecture dependent modules, made by Sweatha Rao and Mitchell Trope, which are specific to our implementation board. They implement the interface to the FPGA as that of a peripheral, by mapping a memory address range to the FPGA resources and then setting the data bus, memory address request, and request type, and forwarding it to the Register Mapper module, point in which the main topic of this thesis project concentrates.

3.3.2.1 FRP Module

The function of the FPGA request processor (FRP) is to separate CPU read and write requests into address ranges.

3.3.2.2 FBA Module

The FPGA Bus Address Block (FBA) module's main function is to decode the request address and length of the data sent by the processor and forward data accordingly.

3.3.2.3 FBD Module

The FPGA Bus Data module (FBD), takes care of forwarding data to and from the CPU and FPGA, and is tightly coupled with the FBA and FRP modules in data timing and signal synchronization.

Once the FPGA memory request is set up, it is sent towards Register Mapper, where all necessary information is sent un a synchronous manner thanks to the timing setup done by the other modules.

3.4 – RTFPGA scheduler model of computation

We followed a structured approach to developing our hardware/software co-designed scheduler module. The approach we followed is outlined by Edward Lee's Model of computation [19]. Following Lee's approach, we defined the scheduler system's ontology (what is a component), epistemology (what knowledge do components share), protocols (how do components communicate), and lexicon (what do components communicate) [19].

1. *Ontology* (components): The ontology of the scheduler is based on defining hardware/software system components. Each component has well defined interfaces to each other, consisting of control and data signals for all inputs and outputs. A change in the data is denoted by a change in a control signal such that each module can execute its corresponding behavior to process the change.

2. *Epistemology* (interfaces): The information that is shared between components will be event time data, event reference pointers, data addresses and interrupts. Components for storage, queue manipulation, scheduling functions, and interrupt generation are defined for use in other modules. The following table lists the interface specification and functions available for all components:

	Interface to modules	**Available functions**
Event	event time data, event reference pointers	Create
Event queue	events, data storage addresses	Add, Delete, Search Event, Reset
Scheduled event	event time data, event reference pointers	Retrieve, Compare
Interrupt	interrupt signal	Create, Receive
Expired event	event time data, event reference pointers	Retrieve, Remove

Figure 3.4 – Component interfaces and functions

3. *Protocol* (communication): Components are shared through the specified communication protocol. Each module continuously polls its control signals at the beginning of every memory clock cycle, and determines a course of action accordingly.

4. *Lexicon* (language of communication): Finally, the lexicon of the system defines an event component. An event is composed of two pieces of information, a 64-bit timestamp specifying when it should be scheduled, and a 32-bit reference pointer to a function that should be run when

scheduled. These pieces of information compromise most of the shared communicated data between system components. Other system information viewed as part of the lexicon includes control signals, generally used as a service request from one module to another module following a chain of command, which will become apparent on the timing diagrams that will follow in the Section 4.

3.5 – RTFPGA Data Flow Chart

The flow of event data throughout the system is hierarchical, but this is difficult to understand from architectural block diagrams of the modules. At the top of the hierarchy are the read and write requests to FPGA-based registers, whose internal workings were described in section 3.3. If these requests are specifically targeting access to the event queue, a new chain of command emerges in which data trickles down the different modules in order to fulfill the request. The flow of information can be shown as a flow chart:

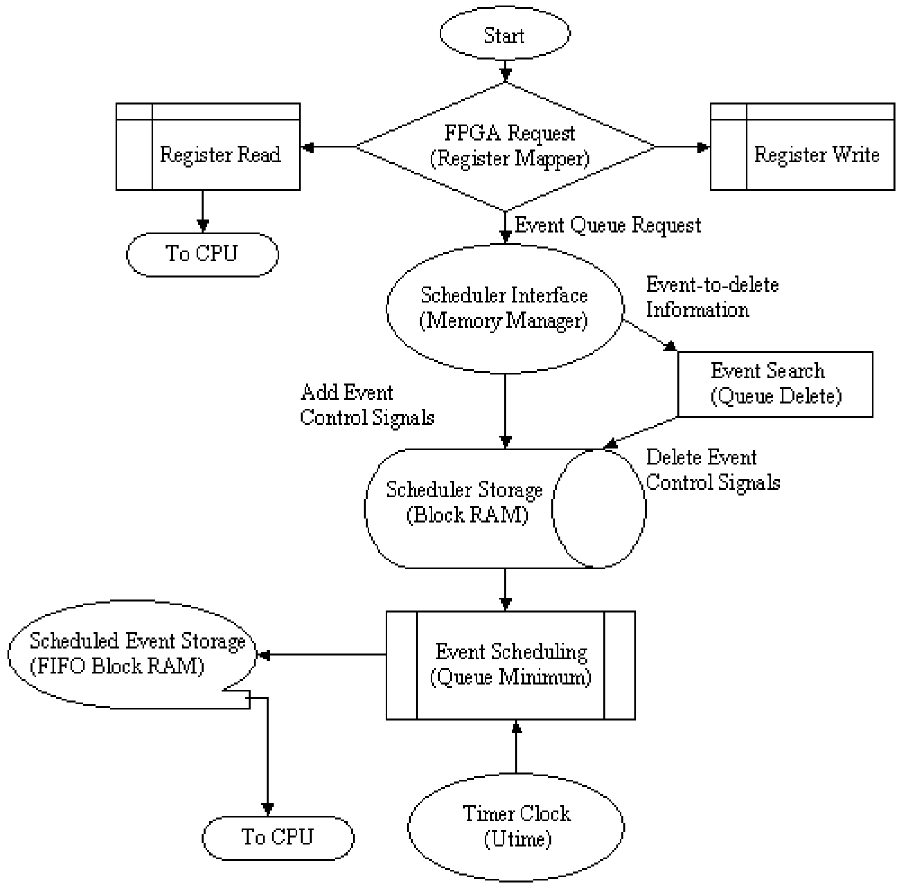

Start
FPGA Request (Register Mapper)
Register Read
Register Write
To CPU
Event Queue Request
Scheduler Interface (Memory Manager)
Event-to-delete Information
Event Search (Queue Delete)
Add Event Control Signals
Delete Event Control Signals
Scheduler Storage (Block RAM)
Scheduled Event Storage (FIFO Block RAM)
Event Scheduling (Queue Minimum)
To CPU
Timer Clock (Utime)

Figure 3.5: Data flow chart in the RTFPGA Scheduler

Event information is passed from the Register Mapper to the Scheduler interface, the Memory Manager module. According to the type of request, suitable control signals are set to execute the command. If we add an event to the queue, the control signals can be set immediately into the Scheduler event storage (Block RAM module), but for a delete request, the specific memory address for the element had to be found through an event search (Queue Delete module), which will then set the control signals.

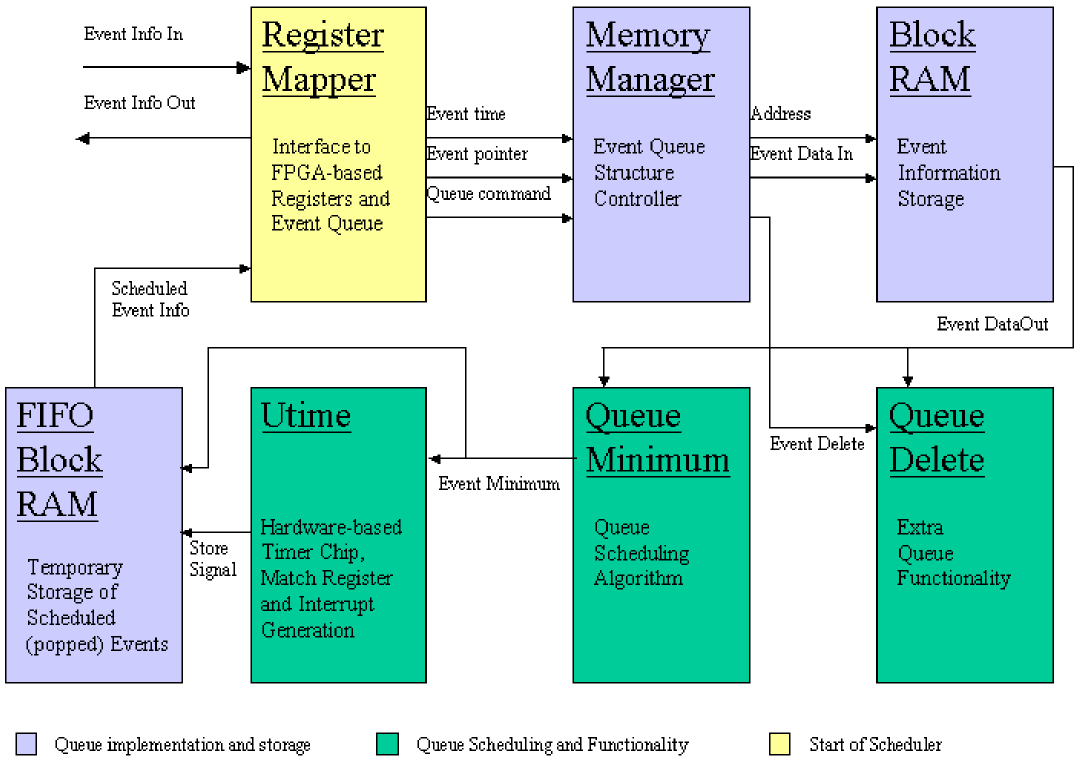
Once the events are stored in the queue, event scheduling starts (in the Queue Minimum module) by receiving data from the event storage and running the scheduling algorithm on this data. We also receive data from the timer clock (Utime module) to check for ready-to-run events that need to be scheduled. When such thing happens, we delete the element from the queue (popping the queue), and save the information for that scheduled event in a secondary piece of memory to be read sequentially by the CPU (FIFO Block RAM module).

3.6 – RTFPGA Scheduler implementation

We implemented the functionality outlined in the data flow chart shown in section 3.5, with the desired system properties from section 3.4, into the existing FPGA modules. These modules integrated the IP cores included with our FPGA board with our additional modules to support our FGPA-based registers, as described in section 3.3.

When a request arrives at the Register Mapper module, we control the address, the data, and the request type. This gives us great flexibility on what we want to achieve with these, since any and all functionality for these FPGA-memory request components are to be implemented in hardware. All further modules in the project stem from the Register Mapper.

For the scheduler's hardware support, several more modules were added, which were outlined in the previous subsection. The layout of these modules can be conceptualized like this:



Event Info In
Event Info Out
Register Mapper
Interface to FPGA-based Registers and Event Queue
Event time
Event pointer
Queue command
Memory Manager
Event Queue Structure Controller
Address
Event Data In
Block RAM
Event Information Storage
Scheduled Event Info
Event DataOut
FIFO Block RAM
Temporary Storage of Scheduled (popped) Events
Store Signal
Utime
Hardware-based Timer Chip, Match Register and Interrupt Generation
Event Minimum
Queue Minimum
Queue Scheduling Algorithm
Event Delete
Queue Delete
Extra Queue Functionality
Queue implementation and storage
Queue Scheduling and Functionality
Start of Scheduler

Figure 3.6: Scheduler modules *concept layout*

The initial modules, Memory Manager and Block RAM, were concerned with the queue implementation and storage. That is, they allocated a portion of FPGA memory, called Block RAM, for use in storing an array of events. Several of these blocks can be instantiated in parallel, allowing arbitrary length reads and writes to be performed.

To allow for a greater level of abstraction, a separate module manages the memory addresses in Block RAM. The Memory Manager module also has an address generator, which drives the data output for use with the queue scheduling and functionality modules. Among its tasks, it keeps count of events, checks for ‘dirty’ addresses in Block RAM, and controls the queue functionality modules.

These scheduling and functionality modules use data coming from stored events to run. These modules, Queue Minimum, Queue Delete and Utime, are color coded accordingly in Figure 3.6 above. The Queue Minimum module keeps track of the next scheduled event by receiving event data from Block RAM and running the Earliest Deadline First algorithm. It then sends the result towards the Utime module. The other queue functionality module is Queue Delete, which takes data from the Memory Manager and compares it to the events in Block RAM for deletion from the queue.

The Utime module is the free-running hardware clock that shadows the clock in the target CPU architecture. In the kernel, this is implemented in the Utime module in the KURT operating system patches. The Utime module receives the next-to-be scheduled event from Queue Minimum, compares it to its internal clock, and upon a match, creates an interrupt to the CPU. Since the CPU cannot read the bus as fast as the FPGA, the event is stored in a FIFO Block RAM queue until the CPU reads it.

We will describe the individual tasks of each module in this section. The next section will concentrate on how all these modules fit together, their interface and inter-module communication, and design data flow.

3.6.1 Register Mapper

The interface to all registers residing on the FPGA is the Register Mapper module. Once a memory request has been routed through all the previously mentioned modules in Section 3.3 (SRP, FRP, FBA, FBD), it is input to Register Mapper. Requests are usually for data, but requests can also be for internal functionality in the FPGA board, like initializing registers or modules, starting counters and sending queue functions like pop and push.

This implementation makes our design quite modular and portable. Once the Register Mapper has been ported to a particular architecture, our scheduler modifications are easily added with minimal modifications. Particular modifications might be needed if a different ‘timestamp’ format is used by a particular architecture.

The Register Mapper is divided into two operational blocks - the read block and the write block. On a read request, the module de-multiplexes the appropriate input from other module data signals into the data bus to the CPU. On a write request, the data bus set up by the CPU is read by the FPGA and the value is multiplexed to the appropriate FPGA module.

The writeable registers within the Register Mapper are:

- initial_jiffy, initial_jiffy_u: *We read the CPU jiffies and jiffies_u, the timekeeping values for the CPU time, add an offset to them, and then write them both them to these register.*
- op_code: *Once the above registers are written to, we write to op_code. This forwards a signal to Utime to start counting in the hardware clock.*
- event_time: *This is the 64-bit register that holds the event time's scheduled jiffy and jiffy_u values.*
- reference_pointer: *32-bit register holding a pointer value to the function to call upon event scheduling.*
- bram_command: *Once the two registers above have been written to, we write a command to the bram_command register, to be serviced by Memory Manager.*

And readable registers are:

- event_time: *This register can be read after being written to.*
- reference_pointer: *This register can be read after being written to.*
- bram_command: *The topmost 24 bits of this register can be used to obtain debugging information, like dirty bits of the queue array, number of events in queue, state machine status, etc.*
- eventtime_from_bram: *this is the value routed from the output of Block RAM after it is written to with a bram_command. Used for FPGA image debugging.*

- refpointer_from_bram: *Same as above. Check for bit flips in FPGA data routing.*
- min_event_time: *This is the value calculated from the scheduling algorithm in Queue Minimum.*
- min_event_ref_ptr_latch: *The values stored in FIFO Block RAM can be read here. Reading this register automatically sends a signal to the FIFO Block RAM to delete the event from the scheduled event queue.*

3.6.2 Memory Manager

The main purpose of Memory Manager is to control the storage and handling of events in the Block RAM-implemented queue. The main inputs for this module are event_time, reference_pointer, and bram_command. The Memory Manager determines which address to write to next, when to run the scheduling function on events, when to perform a linear probe to the queue for a delete, and when to pop the queue upon a scheduled event. It also keeps a dirty bit array indicating which elements in the Block RAM actually have information.

The bram_command register is how the CPU controls what is done with the data. The implemented commands for bram_command are

- Add event: *Stores the data into the next available free address in Block RAM.*
- Delete event: *Takes the events in event_time and reference_pointer for deletion, and starts a loop for linear probing the queue for these values.*

- Pop queue: *When an interrupt is generated in the match register in the Utime module, a signal is sent to Memory Manager to delete the minimum value from the queue, and restart the scheduling algorithm.*
- Reset manager: *This command resets the dirty bit array for taken addresses in Block RAM, and resets other signals for a clean queue.*
- Dirty bits: *This command can output part of the dirty bit array into the bram_command upper bits, where they can be read for debugging.*

3.6.3 Block RAM

This module is an instantiation of the FPGA's Block RAM memory. The Virtex FPGA provides dual-read/write port synchronous Block RAM, with 4096 memory cells. Each port of the Block RAM can be independently configured as a read/write port, a read or a write port, and each port can be configured to a specific data width.

Each port is used for different purposes. The first port will be used to connect to the memory manager requests for writes and reads from the queue. The second port will be used to continuously poll for new values stored inside the block ram. This data flow will be used with two purposes:

1- Run the event priority (earliest deadline) algorithm and

2- Search the queue for events being deleted (e.g., watchdog timers)

3.6.4 Utime

To keep the Linux convention of a jiffy, this module will increment the system heart beat, the jiffies, every 10 ms. To allow for finer scheduling granularity, the jiffies_u will increment at a rate on 1 us to provide real-time microsecond resolution, so this module will start the clock at 100 MHz and update these values accordingly.

Further, the module compares the input data from Queue Minimum to the current system time, and sends out an FIQ interrupt to the CPU, and a timer_off signal to the appropriate modules (Memory Manager, Queue Minimum, FIFO Block RAM).

3.6.5 Queue Minimum

This module implements the Earliest Deadline First algorithm in our scheduler. It polls all the event values stored in Block RAM, by implementing a linear address generator as an independent separate process driving the address input bits in Block RAM. This address generator will make the data output from Block RAM route the event time to a less-than comparator, which will update the current event minimum accordingly.

The pseudo code for this module is shown below:

Pseudo Code for finding earliest deadline event

```
START
SET minimum = max value
LOOP
IF last deleted value = minimum THEN GOTO START
IF last expired event = minimum THEN GOTO START
IF no elements in queue THEN GOTO START
ELSE
     READ new data from Block Ram event queue
     IF data = event (dirty bit is set)
          READ scheduled time
          IF scheduled time < minimum
               minimum = scheduled time
          END IF
     END IF
END IF
GOTO LOOP
```

It does this by initializing the event minimum bits to 1: x"ffffffffffffff". Then it runs the less-than comparison with the received data. We use the dirty bit array input from Memory Manager to distinguish between dirty addresses with event information on them, or empty (or deleted) addresses. This minimum-finding process runs continuously in the background and is stopped only when

1. There are no elements in the queue,
2. The current minimum event has just been scheduled, or
3. There's a deletion process for an element in the queue.

Upon the deletion, it checks if the event to be deleted is the minimum, and resets the value accordingly.

Since there's a non-zero clock cycle delay between the time that the address-generator process sets an address in Block RAM and the time in which the data for the address is routed back to the module, Queue Minimum also sets signals to synchronize data. Since this module has the most up-to-date information on the next event to be scheduled, a separate process implements the interface to the FIFO Block RAM, to add this event to the scheduled event queue immediately after a timer off signal, the signal coming from the match register in Utime, is received. This is done in one clock cycle, and by the next clock cycle, the minimum is reset and the minimum-finding process starts anew.

3.6.6 Queue Delete

By using the address generator from the Queue Minimum and receiving the same event data, we can perform a deletion using linear search. The value being compared against is routed from Memory Manager. If the value matches, the Block RAM address of the event, together with a found_address signal is sent to Memory Manager, which deletes the entry from the queue by unsetting the address' corresponding dirty bit. If a match is not found after checking the Block RAM address range, then a lost_address signal is sent back to Memory Manager and the delete loop stops.

3.6.7 FIFO Block Ram

This module is another instantiation of block RAM, but with a different address controller from the one used in Memory Manager, effectively making it a FIFO queue. The controller for FIFO Block RAM is located in the Queue Minimum module, since this module holds all the relevant information to be stored in the FIFO. The controller keeps track of the head and tail of the queue, and the number of events. Upon a queue pop, the queue_head is incremented (with the circular loop accounted for) and upon a queue push, the queue_tail increments in a similar way, stating which address to write to next.

4. RTFPGA: Detailed Description of Module Communication and Timing Diagrams

4.1 – Module architecture and interfaces

4.1.1 – Event driven layered architecture

The scheduler flowchart was shown in Section 3.5. This flowchart coincides with the layered architecture we envisioned the different hardware modules implementing the scheduler to be. The motivation for using layered architectures is the growing tendency to abstract specifics of the implementation of components from other components or services that do not require that knowledge, only the handling of a specific request. In that sense, the requesting component becomes an upper layer to the servicing component. To increase the level of abstraction and ease of administration, each layer needs to know only two things: which requests it can handle, and know about the layer immediately underneath it to send the requests it can't handle. Its fairly apparent how a layered pattern would be useful for an event driven architecture like ours, were the event is a request from the CPU for services provided by the FPGA-driven backbone of the embedded system.

As specified in the module descriptions, the module will first check its input signals to determine its role on the request. If it is capable of handling the request, it will do so. However, in opposition to common layered architecture practices, the CPU will not receive an explicit acknowledgement back for the completion of the service, which normally would trickle up through the different layers to the requestor. Other CPU commands will be able to check the completion and effects of the request on demand.

4.1.2 – Inter-module interfaces

The interfaces between modules allow for request forwarding including control and data lines. Depending on the type of request being forwarded, the data lines can contain event information like time and reference pointer, or just its address, and hence the data line width is of variable rate. The main interfaces between modules and their respective data line widths are shown in Figure 4.1 below:

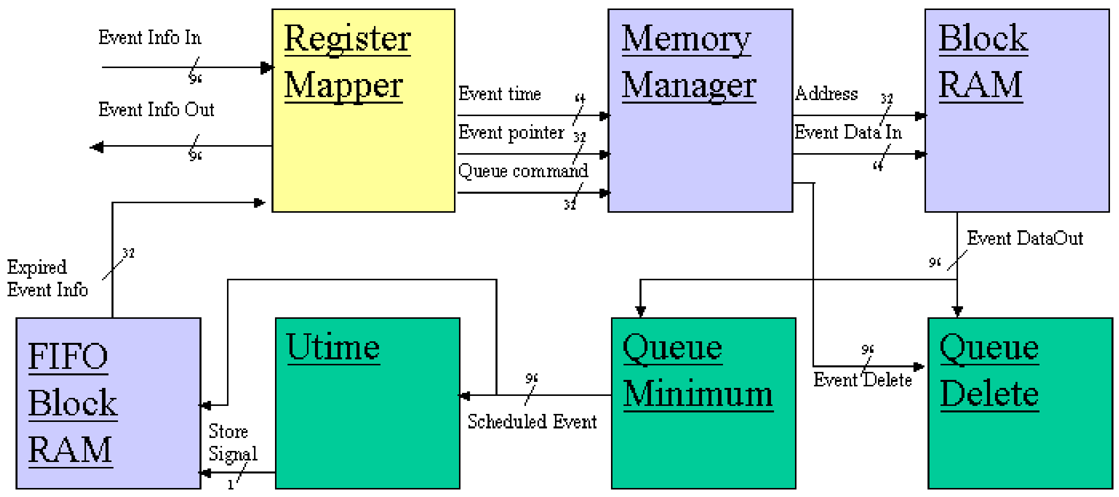


Figure 4.1: Inter-module interfaces

A disadvantage of request forwarding is the amount of data lines that have to be in place between modules, which occupy a large amount of FPGA space, and makes the mapping and routing process for synthesizing the FPGA image harder and prone to delays and errors. Physically, routing resources consume the largest percentage of silicon in many FPGA's. However these resources do not contribute to the computation. [26].

4.2 -Independence and interdependence of entities

4.2.1 – Module independence

Most of the modules do not need interaction to do their specific task; they just need the appropriate input to be feed to them. However, our system is not a pipe-and-filter procedure, and feedback is needed from other modules to stop the functionality and data flow through certain modules to ensure correctness of the system. Among the modules not needing this kind of feedback are the storage elements Block RAM and FIFO Block RAM, and the gateway to the module services, Register Mapper.

4.2.2 – Interdependence of system modules

All the other modules require feedback control signals from other modules to take an appropriate course of action.

The Memory Manager is the most interdependent of other modules. It has to take into account queue deletions happening from scheduled events in the match register in Utime, and deletions happening in Queue Delete. When the Memory Manager requests a deletion to the Queue Delete module, it sets the appropriate signals for this, and waits on the result. Then the Memory Manager receives a signal for the event being found or not in the queue after a linear search. If it is found, it uses the delete address retrieved by Queue Delete to erase that event from the queue dirty bit array and then decrement the number of events in the queue.

Queue Delete will need the appropriate signals set by Memory Manager upon a delete request to produce a correct output for a deletion request. In particular, the scheduling algorithm and the deletion linear search process cannot run at the same time. This is due to the automatic deletion of the minimum event, which happens upon a signal from the match register in the Utime module, when the event is scheduled. Queue Delete has no way of 'knowing' such a deletion has happened due to its reliance on only Memory Manager control signals.

On the other hand, Queue Minimum will also be highly dependant on the Memory Manager module. Not only will Memory Manager order it when to start running, but also when to stop (e.g., when there are no events in the queue or a deletion is taking place), and when to restart (after the current minimum event has just been scheduled and hence erased from the queue). It does this by setting a `minimum_mode` signal, which Queue Minimum checks on every clock cycle to take an appropriate course of action. However, just like Queue Delete needed to be made aware of deletions happening due to expired events in the queue, the scheduling algorithm in Queue Minimum needs to be aware of event deletions talking place. Because the event deletion request is sent to Memory Manager, this module will set the `minimum_mode` signal to stop the scheduling algorithm in Queue Minimum. In specific, when a deletion loop is taking place, the scheduling algorithm will instead check to see if the event being deleted by Memory Manager is indeed the current minimum. If so, Queue Minimum resets its internal values to run again after the delete loop has finished.

4.2.3 – Module cohesion and coupling

The self-referential signals inside Queue Minimum makes this module particularly cohesive. Its internal scheduling algorithm is tightly coupled with other signals the module receives from the exterior, and can be stopped, reset and restarted both by external and internal stimuli. This internal tight coupling is necessary to avoid the use of supplementary modules organizing signals depending on system state. By making all these signals internal to this module, we abstract away the inner workings of the scheduler. However, this does not help in making the scheduling algorithm portable. A change in the scheduling algorithm will need to check for these peculiar (but possible) situations.

The Memory Manager, Queue Minimum and Queue Delete modules have to be, unfortunately, somehow tightly coupled. While loose coupling is always desirable for good engineering, our system is quite specific in its scheduling and timing requirements. Memory Manager is the module with most coupling, since it organizes all queue data. When an event is added, deleted, or scheduled, Memory Manager must know, and also let others know.

4.3 –Timing Diagrams for Inter-module Communication

4.3.1 – Initial setup

Our system is bootstrapped by making the FPGA timer shadow the on the SoC CPU. We do this by reading the timestamp on the CPU, adding an offset, writing this value on the FPGA and finally starting the FPGA timer. The time value is written to the FPGA-based registers initial_jiffy and initial_jiffy_u, which store the values for the initial setup of the hardware clock. After they're stored, the Utime module is signaled to start the clock with these initial values by writing to the op_code register twice: first to load the values from initial_jiffy and initial_jiffy_u from Register Mapper into Utime, and the second one to start the clock with these values. While there might be an unspecified amount of clock cycles between all of the memory requests, we will assume tat they take place one immediately after another. In our timing diagrams, we indicate an unspecified amount of clock cycles by "xxxx".

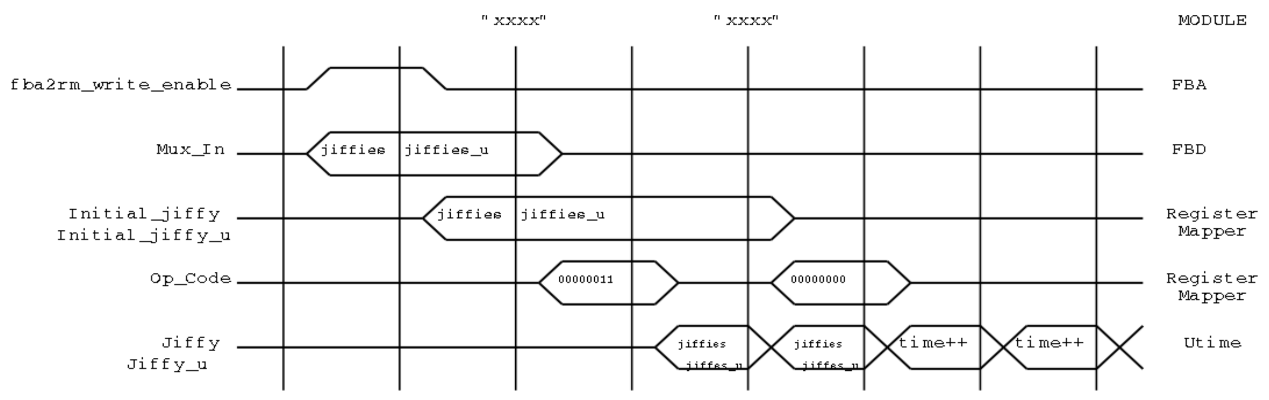


Figure 4.2: Initial setup signal timing diagram

The Mux_In signal is the data bus from the CPU, which gets stored in Register Mapper's Initial_jiffy and Initial_jiffy_u registers when the fba2rm_write_enable signal is set high. Then, when we write the value x"00000011" to the Op_Code register, these time values are forwarded from Register Mapper to the Utime module registers Jiffy and Jiffy_u. This happens every cock cycle until Op_Code is written with any value different from its previous one. After this point, the Utime counter starts incrementing with its starting values as allocated.

4.3.2 – Event Addition Timing Diagram

The write an event, we need 3 write requests from the CPU. Our data bus is only 64-bits wide, so the first one will be writing the event tie, then its reference pointer, and finally writing to the bram_command register to indicate an event addition.

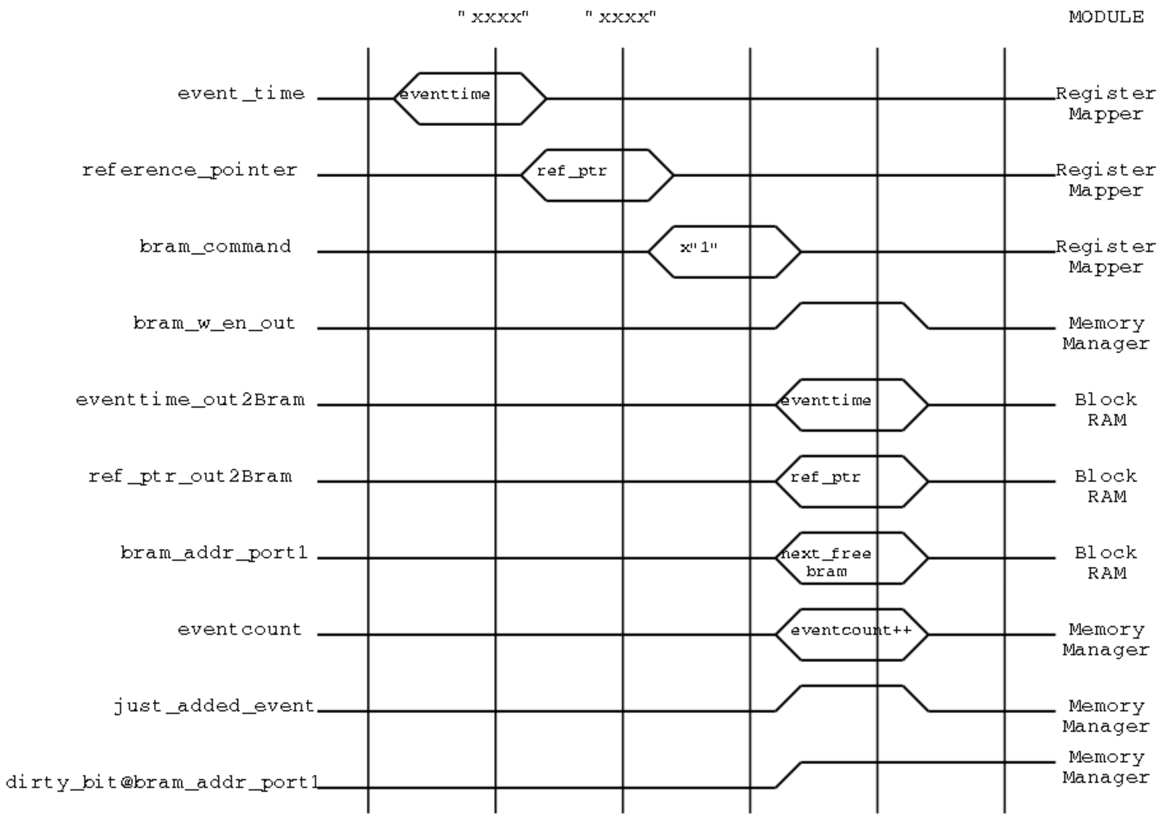


Figure 4.3: Timing diagram for event addition to the queue

Once we write to the event_time and reference_pointer registers, we write the value x"1" into the bram_command register. This is the code for this specific function; command codes for bram_command will be described in later detail in the next section. The Memory Manager module checks the request, and forwards all data to Block RAM. Notice that we can do this in a single clock cycle, since out inter-module interfaces are not limited by bus-length. Then, Memory Manager uses its internal state signals to set up the rest of the request. It sends to Block RAM the address to write to, which was found on a separate process called next_free_addr, running in Memory Manager. It increases the number of events, set the bit in the dirty bit array, and then sends a signal back to the next_free_addr process for it to find a new free location in the Block RAM memory array.

4.3.3 – Event Deletion Timing Diagram

The procedure for deletion is similar as adding an event, except with a different code for bram_command. However, once we make a delete request, it's forwarded to the Queue Delete module, which will start a linear search through the Block RAM looking for the event. Memory Manager will sit idle until a signal back from Queue Delete is received.

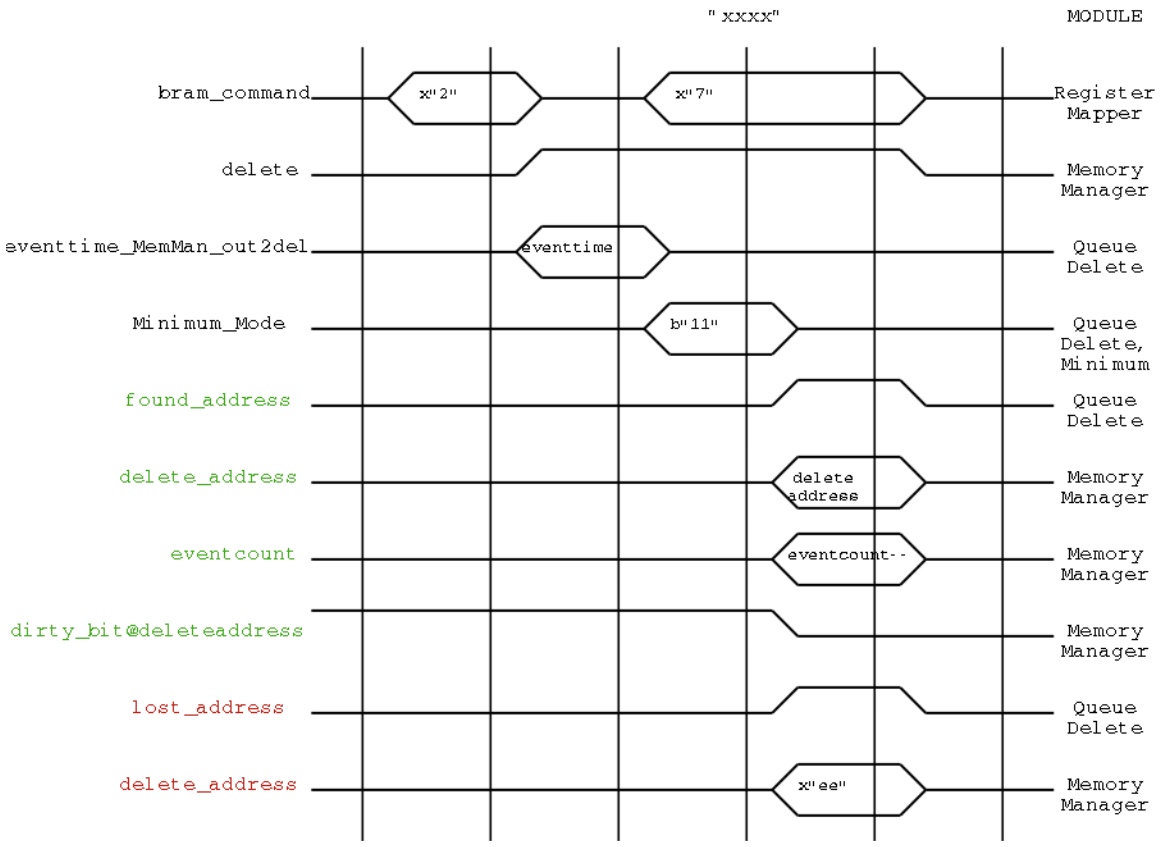


Figure 4.4: Timing diagram for event deletion to the queue

Once this register is written, a delete signal is set, which remains high for the entire course of the delete linear search loop. The event time is sent to the Queue

Delete module, which will start to compare this value to values being received from the always-changing output from Block RAM's secondary port, driven by the address generator in Queue Minimum. When the delete signal is read, the bram_command register is changed accordingly to internally indicate that we are inside the delete loop and should wait for a response from Queue Delete for the service request we made to it. The Minimum Mode is changed to b'11', indicating the delete loop. As indicated in our explanation for interdependence of modules in section 4.2.2, we can't allow the Queue Minimum module to run in parallel with the deletion process in Queue Delete due to concurrency issues. So we set the Minimum Mode to stop Queue Minimum and start Queue Delete. After some amount of clock cycles less than or equal to the queue maximum length, we get a signal back from the Queue Delete module. If it's a found_address signal, we take the address for this event in Block RAM and delete it from there and from the dirty bit array, while decreasing the number of events. If else we do not find the event in the queue, we set a lost_address signal and set the delete_address to x"ee", to let know the CPU that we didn't find the event.

When we receive a signal back from Queue Delete in any of the two cases, we unset the delete signal and reset the bram_command to indicate the end of our delete loop and continue processing requests normally.

4.3.4 – Event Scheduling Timing Diagram

Several things occur in our system when an event is scheduled. We create an FIQ interrupt to inform the CPU that an event needs to run, and we inform appropriate modules about this occurrence, too. Not only do we need to delete the event from the queue, but also notify Queue Minimum that the value from its scheduler is now obsolete. Besides that, we need to queue the event in the FIFO Block RAM for the CPU to read at its own time, but we will also need to check the last time the FIQ signal was set, as the FPGA shouldn't create FIQ signals recklessly, due to the overhead of forcing the CPU to do this. The CPU will receive an FIQ signal only after a threshold time, currently hard-coded to be 30 us. Several signals propagate throughout the module, but all of them stem from the timer_off signal created by the match register in Utime, indicated by red in the diagram timing diagram below:

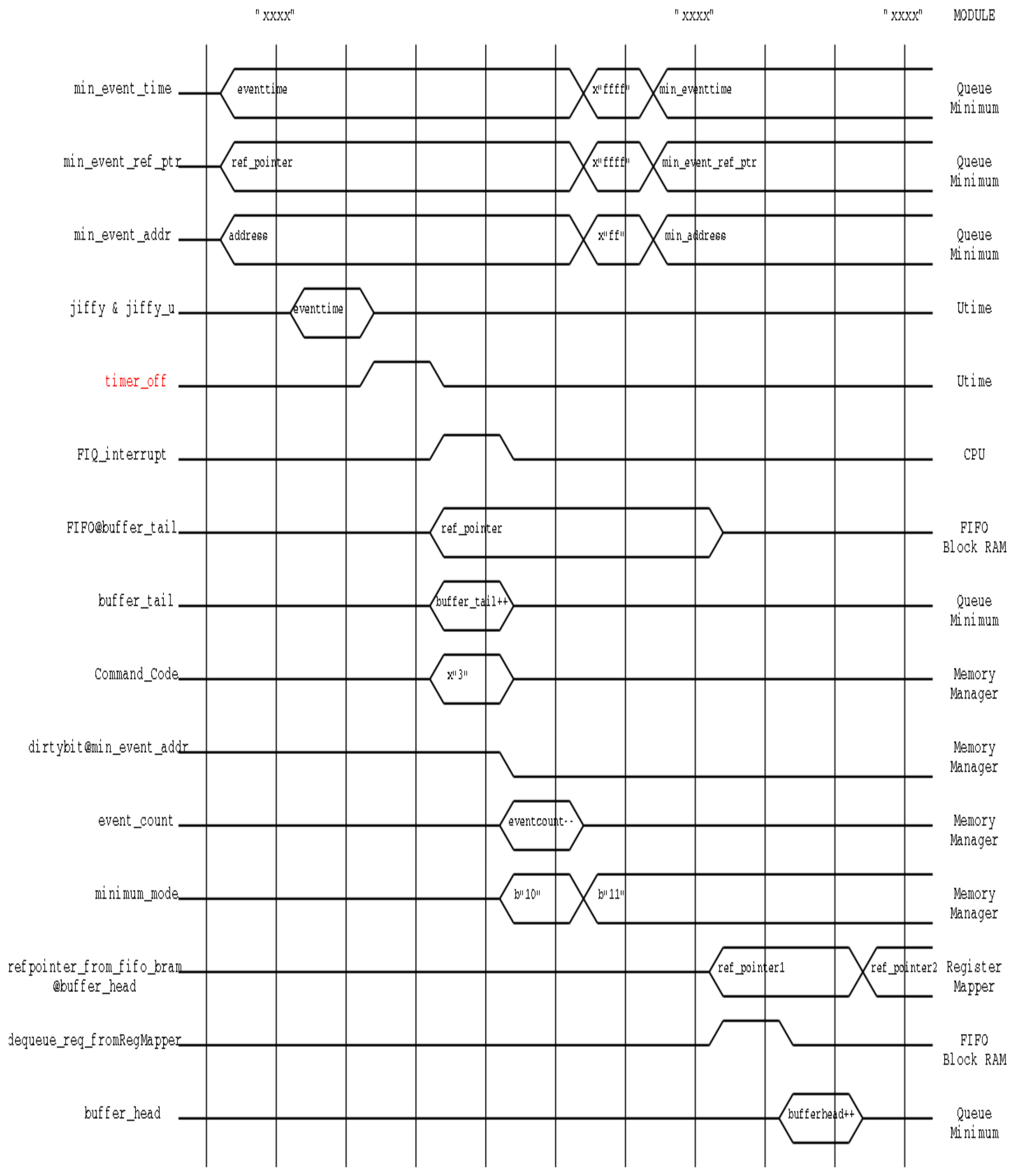


Figure 4.5: Event Scheduling Timing Diagram

Our diagram starts by having Queue Minimum find all the information on our next event to be scheduled. This includes the time, reference pointer and address of the event, which will be used by different modules in the system. When Utime detects that the event time matched the current time from its jiffy and jiffy_u clock registers, it creates a timer_off signal, signal that will be propagated to appropriate modules like Memory Manager, Queue Minimum, and FIFO Block Ram. Depending on previous conditions (FIQ interrupt threshold time, etc), the Utime module sets the FIQ interrupt to the CPU. Our diagram shows it being set for one clock cycle, but in reality the interrupt is set for 50ns, so that the CPU has enough time to sense the signal. Timer_off also acts as a write enable signal for FIFO Block Ram. As soon as it is set, the event reference pointer value coming from Queue Minimum is stored in FIFO Block RAM, and at the same clock cycle the FIFO buffer queue tail is updated since we just pushed an item into the queue. At the same time, the Command Code in Memory Manager changes for it to set all appropriate internal signals and logic in the next clock cycle.

Once this happens, Memory Manager unsets the bit in the dirty bit array for the scheduled event's address, erasing it from the queue and decrementing the number of events. Further, it sets Minimum Mode to "scheduling" for 1 clock cycle. This will indicate the Queue Minimum module to stop and reset the value for the next event to be scheduled, which until now has remained the same. Once this is done, the Minimum Mode goes back to its normal state of continuously finding the EDF event in the queue.

The diagram also shows the effects of the CPU reading the scheduled events in FIFO Block RAM. Such a request if routed through Register Mapper, which detects the read and forwards a de-queue request to the FIFO Block RAM controller in Queue Minimum. This in turn increments the buffer head address to output the next scheduled event that has not been read by the CPU.

5. Simulations and Testing

Once the system was setup, we hooked up the KURT operating system, with DSKI enabled. DSKI stands for Data Streams Kernel Interface. Data streams are representations of events happening at the kernel level, and DSKI provides a general interface for collecting those event traces from the operating system. It is ideal for measurements on real-time system performance and event scheduling accuracy, the reason why we are using it. By running it on KURT, we get a higher resolution for event measurements such as the real-time constraints that we specify when scheduling such events.

The DSKI test will continuously schedule events at a fine resolution. Then it will create a histogram of when the CPU received the events and calculate the difference between the scheduled time and the time the request it was serviced, among other time metrics. The CPU will repeat the same test without using FPGA hardware support, using only the software-based event scheduler manager, which was previously implemented in the system.

5.1 – Non-intrusive monitoring

Using hardware-based tests enables us to have non-intrusive monitoring [37]. This however requires a lot of ‘snooping’ into internally shared data buses between FPGA modules. This would be the ideal situation, however due to constraints on the FPGA area that was being already used by routing, this was not possible. The data bus width for several of the modules was high and frequent enough to cover 75% of the Slice area in our ADI Spartan II FPGA board. For maximum use of high-speed lines between implemented modules, the recommended use of the slice area in a Xilinx FPGA is about 33%. Our need to use more than this amount brought forth several routing and timing delay errors. A system's gate density limits design sizes [36], so we were forced to use only limited amounts of monitoring.

5.2 – Testing Metrics

It is intuitive that one of our metrics would be based on overall performance. However, most of the design was created in order to allow for hardware-based event scheduling, an increase in overall performance would be a most welcomed side effect, or maybe the goal of further research into this project.

The metrics will include the delay experienced for a scheduled event to be serviced from the time it was scheduled. This already is a standard test used by DSKI, but we will be using it with the FPGA support. Several factors come into play for the results of this test, which are accounted for in the next subsection.

The following graph shows the time between when the time an event was scheduled to occur and the time it was actually executed, measured on a Pentium-

200. This was part of the KURT project incentive to increase the temporal granularity of Linux.

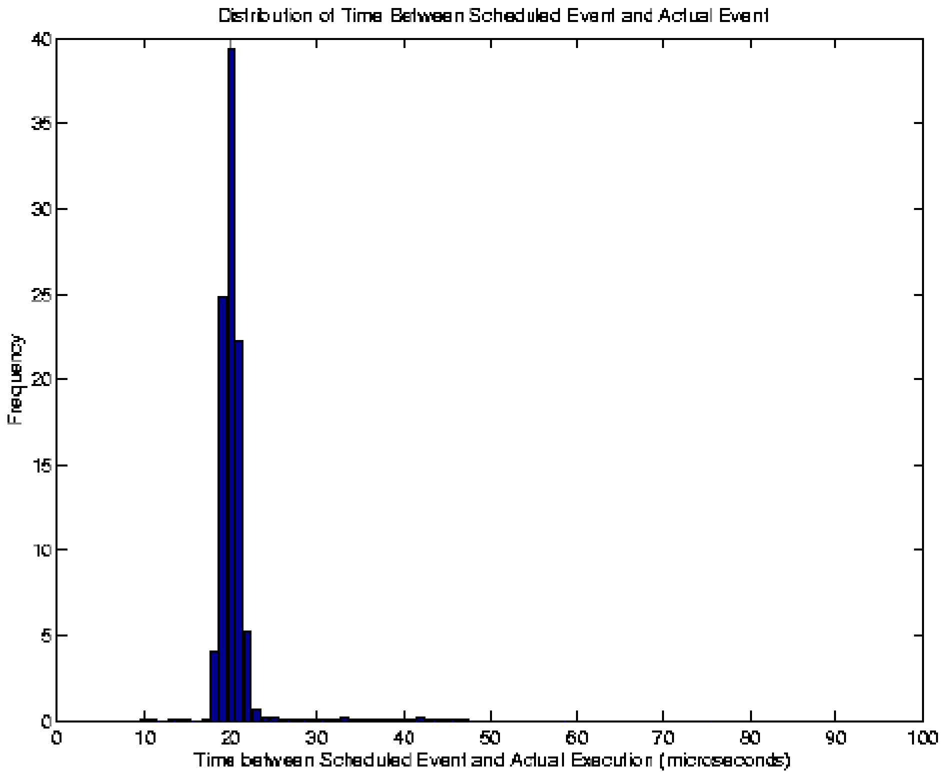


Figure 5.1: Delay between scheduled and execution time without KURT

As we can see, even after the event is correctly scheduled, there’s still overhead time for it to actually be executed. So by letting hardware alone make the scheduling decisions, we shouldn’t expect to see a reduction in the delta time between scheduling and execution of events.

5.3 – Testing Parameters

The hardware-based support for the event queue and scheduling was a faster, bound and predictable approach to solve the issue of handling events. However, it was also dependant on the CPU time availability. It would be futile to performance to rapidly determine the next scheduled event without the need of polling or any other software effort, if the CPU can not read this data early or frequently enough to compensate for this effort.

In our efforts to create a test bench for probing quantifiable and reliable metrics, we found that we had several test parameters that we could not control, and which would contribute to the erratic and sporadic delays that we saw on the CPU-based event scheduling test results. Our reasoning for this behavior includes taking account for the communication costs through the serial port, used for sending data to the Cygmon monitor. Other delays that are also included are random CPU delays due to thread scheduling and context switch costs, serial port communication, CPU delays, FIQ signal delay, etc.

The controllable parameters we had were mostly static throughout all of our tests: jiffy and micro jiffy delays, and event queue depth. The one parameter that we modified in order to contrast the system against its software predecessor, were the number of events scheduled on the DSKI tests.

5.4 – Test Results

5.4.1 – Testing Basic UTIME functionality

The correct behavior of the u-time module was first checked by reading system values (jiffy, micro-jiffy registers), into the CPU and comparing the hardware shadow counter against the software u-time values. This yielded a predictable result of having a constant 'lag' of 2 micro-jiffies in the FPGA. This was credited to the time delay experienced when reading a value from the FPGA, which is considered by the CPU for practical means as a peripheral.

After the initial setup through write requests by the CPU to the timer and control registers, the Utime module starts shadowing the timer value from the CPU in hardware. Iterative reads were made to the timer registers, jiffy and jiffy_u, and the increment of the counters was confirmed. The test for the initial setup is given below.

Initial Set up
Modules: Register Mapper & UTIME

Get system time (cpu_jiffy, cpu_jiffy_u)
Write to initial_jiffy & initial_jiffy_u
Write op_code = 0x00000003(load)
Write op_code = 0x00000005(start)
Read jiffy and jiffy_u iteratively (check for increments)

We modified the Utime module from its final design to drive the FIQ signal high for 5 memory clock cycles at every jiffy boundary (10 ms), for the CPU to detect the interrupt and run the handler. A count of the number of times the FIQ handler ran ascertained that the CPU and the FPGA were in sync with respect to the jiffy count.

5.4.2 – Storing and Deleting Events in the Queue

Since all the event information we need to store is larger than our data bus length, multiples writes are needed to add or delete an event in the queue. Once the processor has provided the event_time and reference_pointer information (associated with a timer) to the FPGA, a third write is made to the control register for addition/deletion. The extra debugging functionality lets the programmer know ahead of time in which position of the BRAM memory an event will be added. Further, any write operation causes the data output port of the BRAM to reflect the data input port. Hence, the value on the output port was latched in a register (eventtime_from_bram) to verify that the write was issued successfully. We also drove the BRAM address to the address we got before adding the event to confirm that the event was indeed added in the expected address. Block Ram addresses, however, are a lower layer of abstraction handled by the Memory Manager module and are only used for debugging purposes. To the CPU, the writing of an event into the queue is transparent.

Some other debugging bits included in the control register allowed us to view the number of events (events_count) in the queue and the addresses at which the events are stored (dirty bits). By tracing back the events added and deleted to the queue and their respective Block ram addresses, we confirmed the correct event queue implementation in hardware. The addition of an event follows the algorithm below

ADD EVENT
Modules: Register Mapper, Memory Manager & BRAM

```
Read bram_command register
IF bram_command(4:7)=x"1" THEN
    Memory Manager is in delete mode, do not issue add request
IF bram_command(8:15)=x"FF" THEN
    The queue is full, do not issue add request.
ELSE
    Write to event_time(0:63)
    Write to ref_ptr(0:31)
    Write x"1" to bram_command(0:3) (add event)
    Read event_time_from_bram (confirm the value written)
    Write x"b" to bram_command(0:3) (events count)
    Read bram_command(16:31) (confirm increment in events count)
END
```

A similar algorithm is run to delete an event from the queue.

DELETE EVENT
Modules: Register Mapper, Memory Manager, Queue Delete, Queue Minimum & BRAM

```
Read bram_command register
IF bram_command(4:7)=x"1" THEN
    Memory Manager is in delete mode, do not issue delete request
IF bram_command(4:7)=x"E" THEN
    The queue is empty, do not issue delete request.
ELSE
    Write to event_time(0:63)
    Write x"2" to bram_command(0:3) (delete event)
    Wait for 750ns (maximum)
    Write x"b" to bram_command(0:3) (events count)
    Read bram_command(16:31) (confirm decrement in events count)
END
```

5.4.3 – Scheduler Event Decision

Once the interrupt recognition by the CPU was ensured, the interrupt was connected to the Queue Minimum module. This module checks every event in the queue and decides on the next event to be scheduled, which is currently the one with the earliest deadline first (EDF algorithm). This event information is forwarded to appropriate modules, including the Utime module. When there's a match between the current time and the event's scheduled time, Utime creates an interrupt signal. This signal propagates through the system creating the FIQ, storing the event info in a buffer, deleting the event from the queue, decreasing the events_count, and setting other control signals. To verify this functionality the following test was run.

SCHEDULER EVENT DESICION
Modules: Register Mapper, Memory Manager, Queue Delete, Queue Minimum, UTIME & BRAM

Add Event1, Add Event2, Add Event3, Add Event4, Add Event5
Read min_event_time (confirm with the events written)
Read min_add (required for popping the timer queue)
Delete Event2 which is has the least event time and check for new minimum values.

5.4.4 – Generation of FIQ and populating the FIFO

As discussed in section 4.3.4 a FIQ cannot be generated for every timer that expires due to software limitations. Hence a FIFO queue was created which stored the reference pointers of all expired timers. When the FIQ is asserted the CPU can initiate a read to the FIFO and schedule the events. A final integrated test was run to verify the functionality of all the components.

INTEGRATED TEST
Modules: Register Mapper, Memory Manager, Queue Delete, Queue Minimum, UTIME, FIFO_BRAM & BRAM

```
1.  Initial Set up
2.  Add Event1, Add Event2, Add Event3, Add Event4, Add Event5
3.  Find Minimum
4.  Delete Event2, Delete Event3
5.  Find Minimum
6.  Read current time from FPGA timer registers (jiffy, jiffy_u)
7.  Write event_time = jiffy + offset
8.  Write ref_ptr = loop variable
9.  Repeat the above three steps for loop variable = 0 to 2
10. Wait on jiffy + large offset
11. WHILE fifo_ref_ptr != x"00000000" (queue not empty)
12. Read fifo_ref_ptr (confirm all the three timers expired)
13. END WHILE
14. Read jiffy & jiffy_u
```

The results of the artificial scenarios created for testing established that all the modules were perfectly integrated and synchronized.

5.5 – Virtex-II Pro platform

The modularity and device independent component design of our system motivated us to port our design from the ADI 80200EVB board to available Virtex-II Pro boards [15] we had for further research. The Virtex–II Pro platform was developed by Xilinx in collaboration with IBM. The important architectural features of this platform are listed below.

- A PowerPC core and programmable logic (FPGA) on the same silicon die providing the advantages of
    1. Reduced area.
    2. Programmable I/O ports in abundance.

3. Ability to create processor-based architectures with required peripherals for any specific application.
4. Reduced system development and debug time.

- A dedicated high performance bus between the processor and it peripherals. This bus is based on the IBM CoreConnect bus architecture consisting of a high-speed PLB (Processor Local Bus) and a general-purpose OPB (On-chip Peripheral Bus). High-performance peripherals such as SDRAM controllers connect to the high-bandwidth, low-latency PLB. Less performance critical peripherals connect to the OPB, reducing traffic on the PLB, which results in greater overall system performance.
- Peripheral IP cores such as UART, Timers, controllers, Fast Ethernet MAC etc, which can be, selected and connected to interface with the high-speed PLB or general-purpose OPB of the CoreConnect bus architecture.
- Programmable logic to implement user defined functions.

The Virtex-II Pro with the embedded PPC core provides flexibility for developing complex embedded systems. Functions can be partitioned between the FPGA logic and the PowerPC core as required by our project.

The porting of our design required less than two weeks, with minimal changes to the platform independent components. The majority of the porting effort was associated with reworking the platform specific interfaces to adhere to the bus architecture and Power PC interrupt structure of the platform [15].

5.5.1 Comparison of the 80200EVB board & Virtex-II Pro platforms

Previous efforts in the ADI 80200EVB board made it possible to get an interface between the FPGA and the processor. This board had an FPGA-based memory decoder for SDRAM and peripheral memory requests. Every memory request was forwarded through this decoder, and while this hampered the modularity of the design, the hard-wired hardware connections to the chip made it possible to have a consistent and predictable behavior for memory requests to FPGA-mapped registers. However, due to a small die area for the FPGA in this chip, we ran into constant place and route problems, mostly involving overuse of Configurable Logic Blocks (CLB) resources in the board, which lead to poor communication lines and bit flipping between the CPU and the FPGA. This caused corruption of data on the bus and improper output for modules.

The Virtex-II Pro offered an architecture in which processor and peripherals communicated with each other over peripheral buses. This enhanced the modularity of the design by eliminating interfaces for the FPGA components. The ready-to-use building blocks and reconfigurable logic provided on the platform eased designing. However, upon running the same tests shown in the next subsection, the price for such flexibility materialized. Requests to the FPGA, together with other peripherals requests, were sent over the OPB bus, managed by a bus arbiter. This accounted for strange bus behavior we didn't expect. While the ADI boards had very deterministic read and write times, the Virtex-II Pro platform had these memory requests subject to varying delay times, even if execute-in-order-execution-of-input-output assembly instructions were used to try and synchronize requests.

5.6 – Data Stream Kernel Interface Tests

The Data Stream Kernel Interface (DSKI) developed at the University of Kansas provides a methodology to evaluate the performance of an operating system. The DSKI user interface enables a user process to describe the set of events it wishes to monitor, thus specifying a data stream customized to its interests [03].

We made use of the HIST_TIMER_EXEC_DELAY test, to compare the performance of the software based RTOS and hardware/software based RTOS. While performance was not our main test objective, it would prove that this functionality could be migrated to hardware, even if there were no expected performance gains. The test programs a new timer after a random time interval and waits for the timer to expire. This process is repeated at every timer interrupt, up to a defined maximum number of timers. Instead of using aggregate measurements for the system, a histogram for the execution delay for events is created.

The results from such histograms can be seen in Figure 6.1. The performance characteristics achieved on the ADI Engineering's 80200EVB board with and without the hardware components of KURT-Linux for a 10,000-event test are displayed. The board functionality also with stood a 100,000-event test.

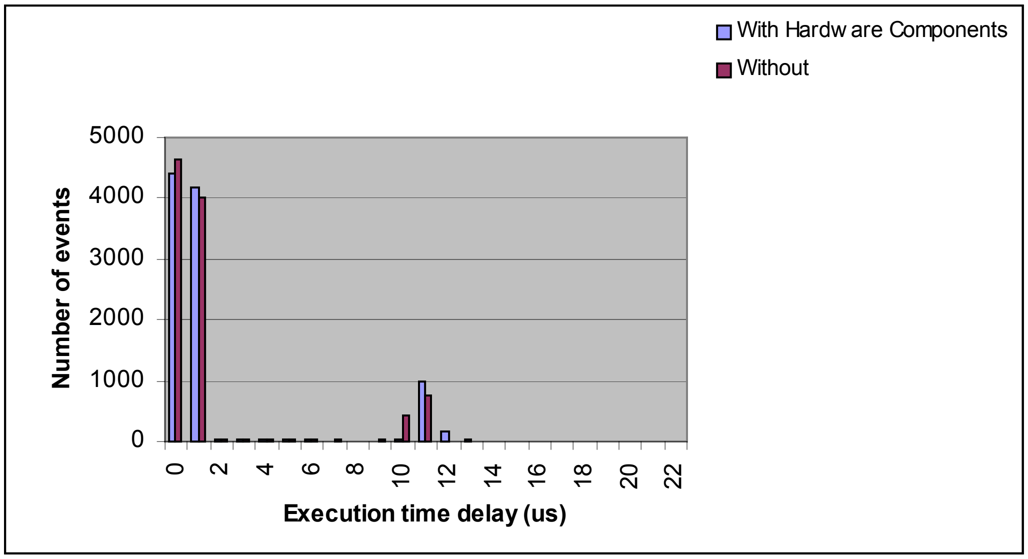


Figure 5.2: Timer Execution Delay for Software RTOS and Hardware/Software RTOS

The Software-based RTOS showed slightly better performance when compared to its corresponding hardware implementation, a bit surprising, but not unexpected. This degradation can be justified through the following reasoning.

- The lag between the CPU and FPGA timers. The timer on the FPGA lags the CPU timer by 2 to 4 microseconds, a difference that is not added when loading the timers on the FPGA. This may be a cause for some events being scheduled later than required. The final test measurements are based on the CPU time, while the interrupts are based on the FPGA timer.
- When an FIQ occurs the FIQ handler schedules an IRQ, downgrading its priority. The FIQ has a higher priority than the IRQ, so an IRQ can be

scheduled but will not be serviced if the previous IRQ context is being serviced when the next FIQ occurs. This may cause delay in the execution of some events.

While no gain in performance was achieved, the hardware implementation provided a more stable time source than its counter software implementation. The result of one of the HIST_TIMER_EXEC_DELAY test on the CPU without any hardware support backs this argument.

```
=========================================================
[root@george dski]$./dskihists -i 900 -f 4 -h 1 -n 100
Waiting for 900 seconds...
 Name of the histogram: HIST_TIMER_EXEC_DELAY
Number of events logged: 10000
Minimum Value: 0
Maximum Value: 7137329
Lower Bound: 0
Upper Bound: 100
Num of Buckets: 102
Range: 1
< 0 is 0
0-0 is 4623
1-1 is 3999
2-2 is 25
3-3 is 29
4-4 is 17
5-5 is 18
6-6 is 19
7-7 is 20
8-8 is 13
9-9 is 29
10-10 is 435
11-11 is 732
12-12 is 5
13-13 is 15
14-14 is 13
15-15 is 0
>= 100 is 7
=========================================================
```

On this test, the maximum execution delay for a timer was in the range of millions of microseconds. This behavior might be attributed to the lack of having an independent timer resource for the processor and was never seen with the Hardware/Software RTOS. In the hardware version, all events were executed within the time delay range of 23 microseconds for all tests.

Another important observation made from our result sets was that there is a spike in the number of events handled between 10 and 11 microseconds from their scheduled time. This was true even for increased or reduced loads in the system. The deviation from the expected exponential decay curve happens due to serial port polling, which makes timer interrupts to be masked during the period of time in which polling happens.

Further, the hardware-supported version yielded better results as the load on the system increased. We used the one test parameter we could manipulate, the number of events to be run, to discover further differences between our software and hardware based schedulers. The software scheduler gave very similar results for different loads, scaled accordingly. However, for the hardware scheduler, when the load was increased from 10K to 100K events, the percentage of events handled within a 0 microsecond delay shot up by 6.4%, and for all events handled within 1 microsecond, 1.7%, pictured in the graph below. The graph also clearly demonstrates a 12.6% increase in percentage of events handled after the serial port polling delay happening between 10 and 11 microseconds after the event scheduled time. Once again, the empirical boundary of 23 microseconds was found to be the maximum delay in hardware for event scheduling, even under heavy loads, which was not the case for the software scheduler.

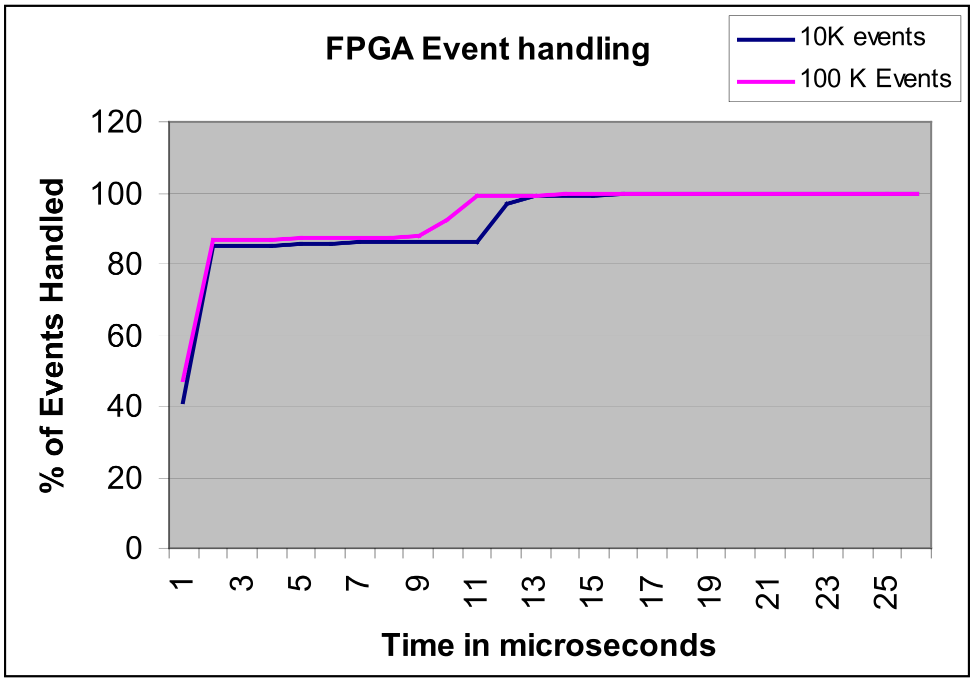


Figure 5.3: FPGA event handling under different event loads

As for non-real time measurements, the aggregate performance characteristics of both systems are comparable. As can be seen from the cumulative timer execution below, most, if not all, of the events are handled within 20 microseconds from their scheduled execution time. This is a pretty robust base indicating the correct and comparable behavior, performance and expectations for our hardware based event scheduler. This will set a strong foundation for current ongoing research projects focusing on moving thread schedulers into hardware.

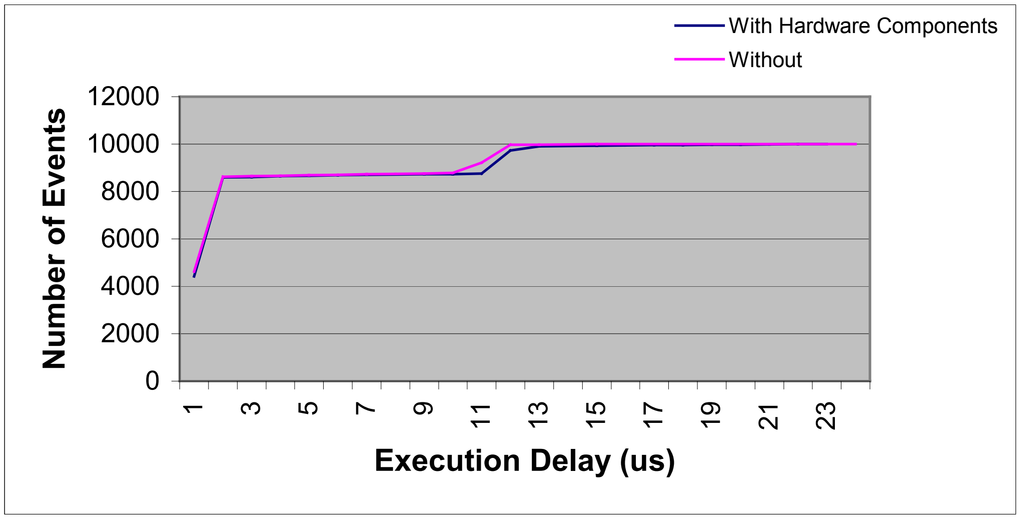


Figure 5.4:Cumulative Timer Execution Delay for Software RTOS and Hardware/Software RTOS

5.7 – System limitations

While hardware wise, there were very few limitations, on its software counterpart the system is limited by several factors. Mainly, the system can only handle interrupts after a certain amount of time, or else we will run into overhead processing problems. Even then, the priority of events is downgraded to keep the interrupts signals enabled. Hence, while on hardware it is possible to schedule several events for the same time, or even within a few microseconds of each other, this does not guarantee that the software will be able to handle such a load accordingly. Hence there's a minimum time delay between scheduled events, which depends on how close together the events were scheduled, and the time spent on servicing one event. Hence, the system granularity, meaning the complexity of handling an event, will have an effect on the system resolution, the number of events that can be handled in a given time period.

For the hardware side, area and non-dynamic allocation of resources limit our system to have a fixed maximum number of stored events (both in the scheduler queue and on the FIFO read queue for expired events), and a constant worst case computation of the next scheduled event of O(N), where N is the maximum number of stored events in the scheduler queue.

6. Conclusions & Future Work

Real time operating systems rely heavily on both the hardware platform the system runs on, the real time services that this platform is able to provide, and the operating system running on the software side. For proper hardware/software co-design of the system, hardware must be utilized to provide timely execution for real time services, while the operating system must grant access to these services with minimum overhead penalties, all while still complying with the constraints of the system. For embedded systems, the increase in complexity of software applications and the real-time services in dedicated hardware are usually mutually exclusive. A more flexible solution was found by combining reconfigurable logic in hardware and a real-time version of a commercially available operating system in software.

KURT-Linux provided a solution for managing real time operations at a finer resolution than was previously available. To reduce the overhead caused by operating system core operations such as time management and event scheduling, the timer and the event scheduler were moved into hardware, which would account for reduced overhead time spent in context switches.

FPGA's provided us with enough flexibility to not only design and implement hardware-based services for real time operating systems, but also to exploit portability by using IP cores when implementing them. This feature was used to test the system in two different platforms, the ADI Engineering's 80200EVB and the Virtex-II Pro. These platform where used to implement the event scheduler.

The hardware-based scheduler showed similar performance to its software counterpart. However, a distinct and important improvement in worst-case scenario was observed when using hardware support. The correctness of the main functions of the event scheduler were tested and evaluated.

While no major performance gain was expected (or achieved) by migrating the event scheduler to hardware, it set a strong base for further research in hardware schedulers. Current research on hardware/software co-design includes creating hardware-based threads [38]. Eventually, these hardware threads will be running in parallel with threads in software on the same system. The scheduler scheduling for these threads will be hardware-based, thus allowing the CPU to be interrupted to make scheduling decisions only when its completely necessary, yielding higher system, processor and resource utilization [39]. At this point, it will be likely that better decision functions will be added to the scheduler, according to system specifications (group scheduling, priority, etc [41]).

7. Bibliography

01. Information and Telecommunication Technology Center (ITTC). *UTIME - Micro-Second Resolution Timers for Linux* http://www.ittc.ku.edu/utime/

02. *KURT-Linux: Kansas University Real-Time Linux* http://www.ittc.ku.edu/kurt/

03. *DSKI - The Data Stream Kernel Interface* http://www.ittc.ku.edu/datastream/

04. Neil Bergmann, Peter Waldeck, John Williams. School of ITEE, University of Queensland. *A Catalog of Hardware Acceleration Techniques for Real-Time Reconfigurable System on Chip* http://eprint.uq.edu.au/archive/00000839/01/iwsoc2003.pdf

05. John A. Stankovic et al, Department of Computer Science, University of Virginia, Charlottesville. *Strategic Directions in Real-Time and Embedded Systems* http://sunset.usc.edu/~neno/cs589_2003/Week5b.ppt

06. Cameron Patterson, Xilinx, Inc. *High Performance DES Encryption in Virtex FPGA's using Jbits*

07. Center for Experimental Research in Computer Systems, Department of Computer Science, Georgia Institute of Technology. *Architecture and Hardware for Scheduling Gigabit Packet Streams* http://www.cc.gatech.edu/~rk/pubs/hoti02.pdf

08. Critical Systems Laboratory, Georgia Institute of Technology. *Architecture and Hardware Support for Real-time Scheduling of Packet Streams* http://www.cc.gatech.edu/~rk/pubs/hpca_ss.pdf

09. Marc Guillemont, Chorus systemes. *Micro kernel Design Yields Real Time* http://www.jaluna.com/developer/papers/CS-TR-90-65.pdf

10. *Linux RT*, Timesys http://www.timesys.com/

11. *RT Linux Toolkit*, FSM Labs http://www.fsmlabs.com/

12. Vincent Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins - IMEC, Belgium. *Designing an Operating System for a Heterogeneous Reconfigurable SoC* http://www.imec.be/reconfigurable/pdf/raw_03_Designing.pdf

13. Steven K. Knapp, Arye Ziklik. Triscend Corporation. *Configurable Embedded Systems* http://www.optimagic.com/acrobat/330.pdf

14. James O. Hamblen, Michael D. Furman. Kluwer Academic Publishers, December 2001. *Rapid Prototyping of Digital Systems*

15. Xilinx. *Virtex-II Pro FPGAS - Lowest System Cost and Highest System Performance* http://direct.xilinx.com/bvdocs/publications/ds083-1.pdf

16. Altera's, *Excalibur Devices Overview*

http://www.altera.com/products/devices/arm/overview/arm-overview.html

17. Ron Sass, Parallel Architecture Research Laboratory, Clemson University. *RCADE - Reconfigurable Application Development Environment*

http://www.parl.clemson.edu/rcc/rcade/index.html

18. Marco Aurelio Antonio Sanvido, *Hardware-in-the-loop Simulation Framework*

http://www-cad.eecs.berkeley.edu/~msanvido/doc/sanvido02phd.pdf

19. Edward A. Lee, IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, May 21-23, 2001. *Computing for Embedded Systems*

http://ptolemy.eecs.berkeley.edu/presentations/01/compEmbSys.pdf

20. Edward A. Lee, University of California, Berkeley. *What's Ahead for Embedded Software?*

http://www.cs.utah.edu/classes/cs6935/papers/lee.pdf

21. Lennart Lindh, Johan Stärner, Johan Furunäs, Joakim Adomat and Mohamed El Shobaki; Mälardalens högskola, Sweden. *Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems*

http://www.mrtc.mdh.se/publications/0114.pdf

22. *SPIN*: On-the-fly, LTL model checking with SPIN

http://spinroot.com/

23. *The Stanford Rapide™ Project*, University of Stanford.

http://pavg.stanford.edu/rapide/

24. *Promela*, Language Reference:

http://spinroot.com/spin/Man/promela.html

25. Javier Resano, Diederik Verkest, Daniel Mozos, Serge Vernalde, Francky Catthoor. *Application of Task Concurrency Management on Dynamically Reconfigurable Hardware Platforms*

http://csdl.computer.org/comp/proceedings/fccm/2003/1979/00/19790278.pdf

26. Barry Ellis Mapen, University of Connecticut. *Efficient Hardware Context Switches in Field Programmable Gate Arrays*

http://www.computing.surrey.ac.uk/personal/st/R.Peel/research/wotug22.pdf

27. *Handel-C*: Language Overview.

http://innovexpo.itee.uq.edu.au/2001/projects/s369358/files1.pdf

28. The Open *SystemC* Initiative

http://www.systemc.org

29. EECS UC Berkeley: Heterogeneous Modeling and Design: *Ptolemy Project*

http://ptolemy.eecs.berkeley.edu

30. G. Vanmeerbeeck P. Schaumont S. Vernalde M. Engels I. Bolsens; Leuven, Belgium. *Hardware - Software Partitioning of embedded system in OCAPI-xl*

http://www.sigda.org/Archives/ProceedingArchives/Codes/Codes2001/papers/2001/codes01/pdffiles/2_1.pdf

31. Massey University: Institute of Information Sciences and Technology: *Correctness of concurrent programs* http://www-ist.massey.ac.nz/csnotes/355/lectures/correctness.pdf

32. Marcus Bednara, Oliver Beyer, Jurgen Teich, Rolf Wanka, Paderborn University, 33095 Paderborn, Germany. *Hardware-Supported Sorting: Design and Tradeoff Analysis*

33. Pramote Kuacharoen, Mohamed A. Shalan and Vincent J. Mooney III, Georgia Institute of Technology. *A Configurable Hardware Scheduler for Real-Time Systems* http://www.ece.gatech.edu/research/codesign/publications/pramote/paper/chs-ERSA03.pdf

34. André DeHon, California Institute of Technology. *Analysis of QuasiStatic Scheduling Techniques in a Virtualized Reconfigurable Machine* http://brass.cs.berkeley.edu/documents/fpga02_sched_print.pdf

35. University of Kansas: Information and Telecommunication Technology Center: *RTFPGA Design and Procedures Documentation*

36. Craig Ulmer, Georgia Institute of Technology. *Configurable Computing - Practical Use of FPGA's*

37. Xilinx. *Get RealFast RTOS with Xilinx FPGA's* http://www.realfast.se/web/info/2003/XILINX_030226.pdf

38. *Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational Components*. David Andrews, Douglas Niehaus, Razali

Jidin; 1st Workshop on Embedded Processor Architectures; Madrid, Spain, Feb. 2004.

39. David Andrews, Douglas Niehaus, and Peter Ashenden. *Programming Models for Hybrid FPGA/CPU Computational Components*. IEEE Computer, January 2004

40. Kiarash Bazargan, Ryan Kastner, Seda Ogrenci, Majid Sarrafzadeh; Northwestern University. *A C to Hardware-Software Compiler* http://www.ece.ucsb.edu/~kastner/papers/C_to_hardware_software_FCCM00.pdf

41. *Distributed Scheduling Aspects for Time-Critical Targeting,* Douglas Niehaus http://www.ittc.ukans.edu/view_project.phtml?id=180

42. Center for Electronic Design Automation, Carnegie Mellon University. *A Codesign Virtual Machine for Hierarchical, Balanced Hardware-Software System Modeling* http://www.sigda.org/Archives/ProceedingArchives/Dac/Dac2000/papers/2000/dac00/pdffiles/23_1.pdf

43. Department of Electrical Engineering, Leuven, Belgium. *A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems* http://www.imec.be/reconfigurable/pdf/prorisc_00_hardware.pdf

44. Department of Computer Science and Engineering, The University of Washington. *RaPiD Coarse-Grained Adaptable Architectures for*

Embedded Systems Platforms
http://www.cs.washington.edu/research/projects/lis/www/rapid/

45. John Wawrzynek, EECS Department University of California, Berkeley. *A Parameterized Library Approach for Specifying and Managing FPGA Computations*
http://www.ucop.edu/research/micro/98_99/98_167.pdf

46. Ted Bapty, Sandeep Neema, Jason Scott, Janos Sztipanovits; Vanderbilt University. *Run-Time Environment for Synthesis of Reconfigurable Computing Systems*
http://www.isis.vanderbilt.edu/projects/acs/Presentations/HPEC_ACS.PDF

47. Antti Pelkonen, VTT Electronics; Kostas Masselos, INTRACOM SA; Miroslav Cupák, IMEC. *System-Level Modeling of Dynamically Reconfigurable Hardware With SystemC*
http://www.ece.lsu.edu/vaidy/raw03/SLIDES/A_Pelkonen.PDF

48. Department of Computer Science and Engineering, The University of Washington. *RaPiD Coarse-Grained Adaptable Architectures for Embedded Systems Platforms*
http://www.cs.washington.edu/research/projects/lis/www/rapid/

49. Herbert Walder and Marco Platzner, Swiss Federal Institute of Technology. *Reconfigurable Hardware Operating Systems*
http://www.tik.ee.ethz.ch/tik/education/lectures/RC/WS03_04/Walder_RCOS.pdf