Bare metal porting of Tasking framework on a Xilinx Board

Gopal Ashish, 2016, AESS, Embedded Systems, ISAE

Dr. Olaf Maibaum, DLR, Braunschweig

Dr. Arnaud Dion, ISAE, Toulouse, France

1 Introduction

The tasking framework is a part of a broader project under DLR called the SCOSA (Scalable On-Board Computing for Space Avionics). As the name suggests the project deals with next generation on board computers for space applications. The tasking framework is an already existing and implemented framework employed by DLR. It is DLR's solution for distributed and parallel computing. The objective of this internship is to achieve the porting of the existing framework on an evaluation board provided by Xilinx.

2 Background

2.1 Tasking Framework

As mentioned in the earlier section the Tasking framework is used for parallel computing. The existing implementations are based upon some operating system running on the hardware. Depending on the hardware available and other project constraints the underlying operating system choice changes and this also accounts for minor changes in the framework itself.

In order to facilitate these changes in the framework with minimum changes from an application programmer's perspective the framework is divided into 2 parts, an API which the application programmer uses to access the tasking framework, and a hardware part which specifies implementation and classes depending on the underlying operating system and hardware choice. As part of this internship the API has been preserved and no changes were made. All changes were made to the hardware section for the bare metal implementation.

The Tasking Framework does not have any computational task of its self. The framework is entrusted with the work of providing access to resources such as memory and CPU time to different sensors and other peripherals. The data from these sensors access request to be processed using the API. Each of such requests is treated as a Task by the framework. Each task comes with an associated priority, and depending on this priority it is queued in list. To accommodate the periodically recurring sensor data the concept of "TaskEvent" is implemented. A TaskEvent can be designed to be a periodic occurrence or a onetime event. TaskEvents are merely data that are pushed at a specific time interval. Each TaskEvent are associated with a task. Every time some new data is available through the Event the associated task is activated and is ready to be scheduled. A TaskEvent and its associated Task has the highest priority among all the tasks and hence are the first to be scheduled.

2.2 Hardware

The work done during this internship revolved around implementing the Tasking framework on a board provided by Xilinx. The board is called the Microzed 7020. It is based on Xilinx's popular Zynq 7000 architecture. The board comes equipped with two ARM Cortex A9 based processors [3] [2] and other peripherals which form the Processing System (PS) part and a Programmable Logic (PL) part. The board provides the flexibility to off load certain amount of computations on the PL from PS by configuring the PL. However in this project the PL is configured with a standard bitstream file generated using the Vivado software to access the other peripherals on the board. The 64 bit global timer, snoop control unit and the Generic interrupt controller (GIC) are among the other prominent resources used as part of this implementation. [1]

3 Development

The existing implementation makes use of the POSIX API to achieve multi threading. Also all the existing implementations were done with an operating system running on the hardware. The initial idea was to achieve an implementation without any OS and without using the POSIX API for threading. All the operating system dependent libraries had to be eliminated and context switching mechanisms had to be investigated to understand how this can be achieved on a bare metal implementation.

All the clocking functionalities are ported to the 64 bit global timer available on the board. Although each processor has an independent 32 bit timer it was decided that the global timer would be a better choice keeping in mind

issues faced in previous implementations. The clock is calibrated to work at 333MHz. The hardware Global timer provides the facility of firing an interrupt with id ID27 every time the timer register values are higher than the comparator register value associated with the timer.[1] This functionality is made use of in implementing TaskEvents in the bare metal implementation of the tasking framework. Every time a TaskEvent request is received the comparator registers of the timer are updated to trigger an interrupt when the time has elapsed. The TaskEvent processing is then implemented in the Interrupt Service Routine. The GIC is configured such that core 0 addresses this Global Timer interrupts. These hardware interrupts have highest priority so it ensures that these are processed first.

Both the cores are configured to work in SMP architecture [6] with core 0 being the master core. The data associated with the application is shared between both the cores by marking a page of the memory as shareable. After boot up the FSBL places the core1 in a Wait for Event (WFE) loop. Core0 is pointed to the application which needs to be executed after start up. Core0 performs all the configurations and environment settings before starting the Core1. We then use the Core0 to wake up the Core1. In order to wake up Core1 we write the address of Core1's application at the address 0XFFFFFFF0 and issue an "sev" command.[1] Upon receiving the event Core1 starts executing the pointed application. In this implementation Core1 executes a loop looking for a task that requires processing. If Core1 is free and available core0 assigns the processing request to Core1 else it is processed by core0 itself. Multiple options were considered for synchronization. Most of the synchronization primitives suggested by example in Microzed Chronicles [4] and ARM mutexes [8] faced cache coherency issues [7]. Finally a compiler intrinsic statement is used to achieve the synchronization of cores. The framework itself is executed by the Core0 in this implementation and the Core1 only executes an executor function. This function receives an activated Task as input and performs the computations associated with this Task. Thus Core0 handles all the aspects of the tasking framework and Core1 is only called upon to share the workload. In a normal condition where there are not many Tasks to be executed at an instant Core1 executes the activated tasks and Core0 keeps the framework active.

4 RESULTS

After having considered multiple implementation options the above mentioned approach was finally adopted where all the Task events are to be scheduled on Core0 as an interrupt and the remaining tasks are distributed among both the cores depending on availability. The implementation was tested for multiple scenarios and emphasis was laid on testing for parallelism. The cores are synchronized while accessing shared resources and parallel behavior was observed when multiple events were fired at the same instant.

5 CONCLUSION

As decided earlier the bare metal implementation was successfully achieved using both the available cores. The synchronization between the cores was a major challenge. Since there are not many documentation regarding SMP in bare metal on the internet ideas were taken from other implementations to achieve synchronization. Some of the more popular synchronization primitives had unexpected behavior when the memory region was not cached. When the memory was cached there were cache coherency issues observed. Finally a compiler intrinsic statement is used to achieve the synchronization between the 2 cores. The possibility to have a block free algorithm was explored, however the algorithm suggested was considered to be under utilization of resources available. This would make a good proposition for future work to consider block free implementations. Also the PL has not been used at all for this project and it would be interesting to consider the possibilities of employing the PL to share the workload.

REFERENCES

[1] Technical Manual of Zynq 7000 All programmable SoC http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
[2] ARM-Cortex A9 MP-core Technical Reference Manual http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.p df
[3] ARM-Cortex A9 Technical Reference Manual http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf
[4] Microzed Chronicles by Adam Taylor Website:http://zedboard.org/content/microzed-chronicles
[5] Zedboard support forums http://zedboard.org/forums/zed-english-forum
[6] AMP and SMP http://rtcmagazine.com/articles/view/101663
[7] Cache coherency description http://www.webopedia.com/TERM/C/cache_coherence.html
[8] Assembly Mutex http://stackoverflow.com/questions/29783951/can-i-use-ldrex-strex-to-implement-a-spin-lock-without-enabling-scu-in-a-multico
[9] Vivado support https://forums.xilinx.com/t5/Embedded-Development-Tools/XAPP1079-in-Vivado-2015-4/td-p/678257