Co-processor for Microkernel OS Services

Verber Domen

University of Maribor, Faculty of Electrical Engineering and Computer Sciences, Maribor, Slovenia (e-mail: domen.verber@uni-mb.si)

Abstract: Traditionally the operating systems are implemented in software. This yields to larger memory footprint, introduces interference in normal operation of the application, lengthens the reaction times, reduces throughput and makes temporal analysis of the system more difficult. To eliminate this, a co-processor that performs the functions of the operating system can be employed. In the paper an implementation of a full-featured co-processor for microkernel operating systems services is presented. This solution virtually eliminates the impact of the operating systems to the normal application execution and outperforms any software implementation. The prototype of such co-processor implemented with FPGA devices has been built and evaluated.

Keywords: Operating systems, embedded systems, real-time, task scheduling, FPGA.

1. INTRODUCTION

Operating systems (OS) are integral part for almost all but the simplest of control systems. For a control application, OS provides the abstract interface to the hardware and implements some services like task scheduling, task synchronization, inter-task communication, etc. In the paper, a proposition for implementing some of the OS functionalities by hardware means is presented. Theoretical studies were augmented with a prototype built with the FPGA device.

The usage of OS co-processor can be considered the same way as the mathematical co-processor. The first generation of microprocessors was not capable to perform floating-point mathematical operations directly. Instead, software routines were used. However, the execution times of such mathematical operations were for a magnitude slower than the equivalent integer operations. To deal with this, manufactures of the microprocessors started to implement math co-processors. Later on, with the advances of technology, the math co-processors became an integral part of the general processors. The similar scenario may be employed for the OS co-processors. Today, operating system is implemented in software. Microprocessors provide only rudimentary support for the OS (e.g. limited support for context-switch implementation, atomic instructions for synchronizers implementation, etc.). As such the OS consumes considerable amount of memory which can be problematic for an embedded system. The OS routines have prolonged execution times. In real-time systems, they interfere with normal operation of the application and make temporal behaviour analysis very difficult. With OS implemented in hardware, the memory footprint is much smaller, the execution time of OS services could be near zero and the impact on the control application would be much smaller. In the future, OS co-processors may be integrated into the processor core.

There has been some previous research done on the implementation of the OS functionality in hardware. However, most of it was focused only on certain parts of the OS functionality, mainly the task scheduling (examples Halang (1988), Lindh (1989), Cooling (1993), etc.). At the time when these ideas were presented, the technology of complex custom devices was not mature enough to be practical and successful. In late 1980’s the manufacturer Inmos introduced the Transputer. This was a microprocessor with integrated task scheduling support. However, the architecture of the Transputer was difficult to be used with a broad range of applications at that time. The Transputer is not produced anymore. At the same time, several hardware implementations of the Japan’s TRON (The Real-Time Operating Nucleus) operating systems were proposed (e.g., Silicon TRON, Nakano, et al., (1995)).

Recently, as the prices of silicon drop, a boost on hardware implementations of different software algorithms can be observed. Again, different parts of the operating system were co-designed in hardware and software (Vetromille, et al., (2006)). In addition to basic kernel operations, more complex operating system functionalities have also been implemented in hardware (e.g., communication support, Maruyama, et al. (2010)).

In our previous research, we tried to implement earliest deadline first (EDF) scheduling algorithm for hard real-time systems in hardware (Verber (2009)). It ensures that all task scheduling operation may be executed in constant time (i.e., with O(1) time complexity). However, to make this possible, we found that all tasking operations must be taken into account. When we have built a prototype, we noticed that it could be extended to the other OS services. Furthermore, since EDF scheduling is such a complex algorithm, we concluded that it would be even easier to implement other scheduling strategies. This would make the OS co-processor much more usable for various kinds of embedded control systems.
In comparison to most other hardware implementations, we were capable to achieve the execution time of maximum four clock cycles for every co-processor operation.

In the next section, a brief descriptions of OS services that are feasible for hardware implementation is discussed. After that some more important issues of hardware implementations are given. At the end, the practical results of experiments are presented.

2. BASIC MICROKERNEL OS SERVICES

By the term microkernel usually a fundamental set of OS services is meant. In the article, these OS services are described only briefly to facilitate later discussion of the implementation. For more details, please, consider other resources (e.g., Silberschatz et al. (2009)). There are some other services usually implemented within the OS. For example, the OS may perform some sort of memory management, it may be used to support the usage of complex input/output devices, mass memory storage devices, user interfaces, etc. However, these services are not usually considered as a part of an embedded application or as a part of a microkernel. Nevertheless, some of these services are also suitable candidates for the hardware implementation and we plan to study them in the future.

2.1 Task scheduling

Most applications for embedded control systems consist of several tasks that are seemingly executing in parallel. Because there is usually only one processing element (or a limited set of them), the tasks must be somehow scheduled to use processing resources effectively. For this, OS keeps the track of all tasks in the systems and determines which one of them will be executed next (using the processor). There is a so-called task control block or TCB maintained for each task by the OS. The TCB consists of a task identity number, the current operational state of the task (i.e., if it is active or not), the working set of internal registers for the task (the context), etc. The employed scheduling strategy depends on the nature of the control application. However, what all scheduling strategies have in common is that there are always tasks that have precedence (or priority) in execution before the others. If a task with the higher priority than the priority of currently running task becomes operational, the execution of later must be temporarily suspended and the first will start the execution. During that procedure, the working state of one task must be saved and the working state of another must be restored. This operation (so-called context-switch or task pre-emption) is ideally performed in zero time. To facilitate the scheduling, the OS usually maintains a sorted list of tasks that are currently prepared for the execution. The tasks with the higher priority are kept on top of a list. The first task in the list is the one that is currently executing on the processor. Alternatively, there may be several queues of tasks associated with each priority level. A task may become active on program request or due to some event in the system. In addition, it may be activated periodically.

The basic task scheduling strategy is the so-called fixed priority scheduling where the priorities of the task are set by the programmers in advance. The number of priority levels is usually kept relatively small. If there are several tasks with the same priority, they are scheduled in first-come-first-served way. In addition, there may also be a so-called time-sharing, where processing time is divided equally among the tasks. Each task has the opportunity to execute for a single time frame (time slice), then another task takes the processor. Time-sharing is usually performed in round-robin fashion: the task that was using the current time frame is put at the end of the waiting queue. It will get the next time frame only after all other tasks with the same priority get one.

Assigning priorities is a difficult task with little guarantee that it was done properly. Alternatively, the priority may be determined as a part of the scheduling strategy itself. For example, for real-time applications the scheduling strategies must be used where execution time of tasks and the deadlines must be considered. Usually, a task with the nearest deadline has the precedence from the others. For periodic tasks, a so-called rate monotonic scheduling is usually employed. There the tasks are prioritized according to its periods. A task with the shorter period has precedence upon the tasks with the longer. Another real-time scheduling strategy is the so-called earliest deadline first (EDF). In this case, the task’s priority is corresponding with its deadline. Shorter deadline means higher priority.

For hard real-time system, the so-called schedulability test must also be performed. With this, we prove that all currently scheduled tasks will finish the execution prior its deadline. The schedulability test is a complex and time-consuming operation.

Sometimes it is necessary that the task, which is currently running, continues its execution even if there is a task with a higher priority. In these so-called critical sections, the OS temporarily disables the scheduling.

2.2 Task synchronization

Apart from the processor, there may be some other resources that must be shared among the tasks (e.g. some unique output device). When some task seize such resource, the others must wait. To facilitate this, the OS implements some sort of synchronization mechanism (e.g., mutex, semaphore, etc.). With software implementation, the OS usually maintains a queue of waiting tasks for each synchronizer. When a resource becomes available, the first task in the waiting queue seize it.

A task can also be temporarily suspended programatically and remains suspended until some event occurs (e.g., a stimulus is acquired from the controlling environment, a signal is sent by another task, a timer runs out, etc.). The task is waken-up automatically by the OS. The application may also trigger the continuation of the task execution explicitly by OS system call.
2.3 Inter-task communication

To cooperate, tasks may need to communicate with each other. For that, there are several mechanisms supported by different OS.

In the case of so-called shared memory, several tasks are allowed to access the same area of memory. To avoid conflicts with mutual changing of data, synchronization mechanisms must be used. Shared memory may be reduced to a set of shared variables.

Other kinds of inter-task communication are message-passing mechanisms. A task may send a message to another task. The messages can be kept in the queue and consumed as necessary. The receiving task may also be set for activation upon the arrival of the message. The most primitive kind of messages are signals that can be used for task synchronizations.

2.4 Time keeping

In real-time systems, a proper consideration of time is important. The OS usually maintains an absolute time clock. It also provides a set of timers for event generation.

Some OS provide also a watchdog facility. The watchdog timer is reset periodically by the application. If it runs out, the alarm is raised to signal the misbehaviour of the program.

3. IMPLEMENTATION OF MICROKERNEL IN HARDWARE

3.1 General process of OS service execution

The block diagram of the co-processor is presented in Fig. 1.

![Conceptual diagram of the kernel co-processor](image)

Fig. 1. Conceptual diagram of the kernel co-processor

The microprocessor interface connects the co-processor to the main processor through the address and data buses. It is exposed to the main processor as a set of registers and memory locations. Some OS services are issued by the main processor by writing an instruction code and parameters into the co-processor. After the operation is finished, the results may be read from the appropriate output registers. Other OS services are provided directly with dedicated logic and are executed immediately by accessing the co-processor's memory locations. Examples are: the setting of control registers, usage of shared variables, etc. We named such operations pseudo instructions.

Some operations may be initiated autonomously by the co-processor itself. To notify the main processor that some event occurs (e.g., the context-switch is required), the co-processor may trigger an interrupt.

The execution units execute one of the co-processor’s instructions. The instructions are executed by means of different sub components described later. For the instruction execution, several sub components may operate in accordance and in parallel.

Which instruction will be executed is determined by the instruction dispatch unit. There are several instruction register sets; each may contain a different OS instruction. The content of each register set is described on Fig. 2. Most registers in the instruction register set are 8 bits in size. The Priority and ExecTime registers are 16 bits long.

Not all registers are used with each OS operation. The complete list of OS service instructions currently supported by the co-processor prototype is presented in Appendix A.

![Instruction register set](image)

Fig. 2. Instruction register set

The OSOper register contains the operation code for OS service to be executed. The TaskID is the register that represents the task identification number for most of the operations.

The Priority register corresponds to the priority of the task at task activation. The actual meaning of the content of this register is dependent on current scheduling algorithm used by the co-processors. For the fixed priority scheduling, this is the actual priority of the task. However, to make the implementation of different scheduling strategies as simple as possible, a lower priority number corresponds with the higher priority of the task. For rate-monotonic scheduling, the priority is the repetition period of a task. Shorter period means higher priority. For EDF scheduling value of the priority is set by the relative deadline of a task. A task with a nearest deadline has the highest priority.
The \textit{ExecTime} register is used for real-time scheduling and represents the total execution time of a task at activation. This value is used later on to perform the schedulability tests and to alarm the main processor that the actual execution time of tasks is longer than expected.

The instruction issue control register (\textit{IssueCtrl}) determines at what condition the instruction will be dispatched to the execution units. For this purpose, one of the event generators may be used. Several instruction register sets may be connected to the same event generator. Each event generator has its corresponding set of control registers exposed directly to the main processor.

There are several kinds of events generators. A simple one is event generator that is connected directly to some external events input. This allows instruction dispatching upon the arrival of a certain signal from the controlled environment. The external events can be masked by means of control register. In a similar way, several external events can be or-ed together to produce the same event.

Another event generator is used for timed operations. The event can be generated only once after some time, or it can be generated periodically (indefinitely or until some point in time).

An event can be generated also by the shared variables described later. When a new value is written into a specific shared variable, an event may be triggered. This is implemented by observing the write operations made into the addressing space of shared variables. In our earlier experiments, we also implemented the triggering of events in the case if the value of the variable is out of some predetermined range.

The instruction register set at position 0 is used for immediate instruction execution. It dispatches the instruction as soon as the \textit{OSOper} register is written into and it does not contain \textit{IssueCtrl} register. Other instruction register sets can also be configured in such way. With them, the main processor may prepare the parameters for some instructions in advance and then issue the command instantly with a single write operation.

There is an extra instruction register set used in the case of time-shared scheduling. With this, the \textit{TIMESLICE} operation is issued periodically to change the task that is currently using the time-slice.

### 3.2 Task scheduling

The largest of the execution units is responsible for task scheduling. It is shown on Fig. 3.

The task scheduling execution unit consists of interconnected cells arranged in a linked list. Each cell contains a set of registers that corresponds to the information held in a TCB for a single task. There are task identity number, priority and task state.

![Fig. 3. Task scheduling execution unit.](image)

The cells in the list are sorted according to their priority values. The contents of each cell can be copied to the previous or to the subsequent cell. The contents of several cells can be shifted in one direction or another simultaneously. Each cell also contains all the logic necessary for the execution of different OS operations.

By exploiting the parallelism, it was possible to implement the task scheduling in such way that each tasking operation takes only four clock cycles regardless of the number of cells. For example, when a new task becomes active (an \textit{ACTIVATION} operation is issued), first, the scheduler finds a proper place for the new data in the list. This is done by simultaneously comparing the priority of the new task with the priorities of the existing tasks in the list. All cells with the greater priority are marked. Secondly, the content of the marked cells is shifted toward the end of the list to provide space for the new task. In the next cycle, the new data is written into the provided place. At the same time, the schedulability test is performed for real-time scheduling cases. In the fourth clock cycle, the scheduler determines the task that must be executed next. It writes its identity number in the \textit{CurTID} register. Other operations are performed in similar way.

Although some operations may be executed faster than in four clock cycles, we found that the implementation is more effective if we always use all cycles. For example, it is easier to implement that the next task to be executed is determined in the fourth clock cycle and leave some other cycles idle.

For real-time scheduling, each cell in the list also maintains two values: remaining execution time of the current task and cumulative remaining execution times of all tasks before and including the current one. The later must be always smaller or same as the current deadline of the task. Those values are updated periodically by the \textit{TICK} instruction. This instruction may be issued by the main processor or it can be performed automatically. For this, no extra instruction register set is required.

### 3.3 Task synchronization

For task synchronization, simple binary semaphores have been implemented. However, it would be easy to expand them into more sophisticated ones. All semaphores are
controlled with separate execution units responsible for the execution of LOCK/UNLOCK operations. Maintaining a separate queue of waiting task for a synchronizer would not be feasible for hardware implementation. Instead of this, the set of synchronizers that a specific task is waiting for is kept in each cell of the scheduler. This is implemented as a set of bits. Each bit corresponds to a semaphore or another synchronization resource. If any bit in this set is different than zero, the task is suspended. Task scheduler ignores the suspended tasks during scheduling.

Another way to suspend a task is explicitly with SUSPEND operation. The corresponding bit is cleared with the CONTINUE instruction.

A task can also be suspended until some shared variable is written into. By this, it is possible to implement simple Wait-and-Signal synchronization methods.

### 3.4 Inter task communication

Inter task communication is implemented with the shared variables. A set of small number of 16 bit shared variables is exposed directly to the main processor. As mentioned before, an event generator may be configured to trigger the execution of the OS operation.

We also experiment with other kinds of inter task communication techniques. It would be easy to implement mailboxes, however only with the message-waiting queue of fixed size.

### 3.5 Other instructions

The prototype of co-processor also implements the watchdog facility and the real time clock, although the later must be set by the main processor at start-up of the application.

For debugging purposes, the microprocessor interface also provides access to the most internal registers.

### 4. RESULT OF THE EXPERIMENTS

The prototype of the co-processor has been built with the Xilinx FPGA device Spartan2E xc2s300e running at 50 Mhz (Xilinx (2010)). For the implementation, the ISE 7.1 tool was used. This specific FPGA device is nowadays obsolete. However, the silicon consumption of the newest devices is roughly the same. The main processor was Texas Instruments’ digital signal processor TMS320C6771 running at 150 Mhz (Texas Instruments (2010)). For testing purposes, a pseudo OS has been implemented that uses co-processor routines. For tasks, we used native task support provided by the operating system of the main processor. The bodies of the task were synthetic and consisted of OS instructions with delays between them.

Several configurations have been tested. First, we have tested implementation of different scheduling algorithms separately. For each model we used eight task scheduling cells, four synchronizers an 16 shared variables. For rate-monotonic scheduling eight dispatching units were used instead of four in other cases. The schedulability test was implemented only in the case of the EDF scheduling.

At the end, we constructed the final model where all scheduling methods were implemented together and the main processor could choose which one should be used. In this model, it is also possible, for example, to combine time-sharing with the rate monotonic scheduling, or to perform schedulability test on scheduling by other scheduling methods than EDF with no extra costs on silicon.

We measured the silicon consumption of each model and the relative results are given in Table 1. The simplest case with the fixed-priority based scheduling was chosen as the unit. All other models are compared relatively to this one. EDF scheduling consumes most of the silicon because of the complex implementation of the schedulability test. The consumption of silicon for the combined model is not much higher because same components are used with different scheduling methods. In absolute terms, the fixed-priority based scheduling requires 1080 programmable logical blocks (slices). The absolute silicon consumption on the specific FPGA device was about 35% in the simplest case and 75% in the combined case. In general, the newest FPGA devices can provide much more programmable logical blocks. For comparison, we simulated the design on a more sophisticated device Spartan3A xc3s3d400A. Here the absolute numbers of slices consumed were 1167. This represents only 4% of available silicon area.

<table>
<thead>
<tr>
<th>Scheduling method</th>
<th>Silicon consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority based</td>
<td>1</td>
</tr>
<tr>
<td>Priority based with time-sharing</td>
<td>1.1</td>
</tr>
<tr>
<td>Rate-monotonic</td>
<td>1.4</td>
</tr>
<tr>
<td>EDF</td>
<td>1.9</td>
</tr>
<tr>
<td>Combined</td>
<td>2.2</td>
</tr>
</tbody>
</table>

### 5. CONCLUSION

At the current state of technology, the hardware implementation of software algorithms is feasible even for simple embedded system solutions. Operating systems have well defined and relatively limited set of functionalities. Therefore, it is easy to imagine an “OS-on-a-chip” solution that may be used in the same way as mathematical co-processors two decades ago or as graphical co-processors are used today. They may even become a part of general processors in the future.

The solution presented has the benefit to provide small memory footprint of the software part, much better response time of the OS and much less interference with the normal operation of the application. On the negative side, the parameters of the OS (e.g., maximum number of tasks in the system) are set in advance and cannot be changed afterwards. In addition, extra silicon usage may increase power consumption and heat dissipation that may be problematic for some applications.
Some modern microprocessors have already supported the so-called user defined instructions (UDI) and associated coprocessor interface. This allows for customized extension of the microprocessor's instruction set with different addressing modes. A study to utilize such interface for the proposed solution is under development. In the ongoing research, the hardware implementation of other functionalities of the operating systems and the communication services is studied. A realistic test application is planned.

REFERENCES


D. Verber. Attaining EDF Task Scheduling with O(1) Time Complexity, Proceedings of 7th Workshop on Advanced Control and Diagnosis, November 2009, Zielona Gora, Poland.


N. Maruyama, T. Ishihara, and H. Yasuura. An RTOS in hardware for energy efficient software-based TCP/IP processing. IEEE 8th Symposium on Application Specific Processors (SASP), 2010


Appendix A. CO-PROCESSOR INSTRUCTION SET

Instructions executed by the execution units

<table>
<thead>
<tr>
<th>Co-processor Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE &lt;tid&gt;,&lt;pro&gt;...</td>
<td>Activation of task. The parameters are task identification number and the initial priority. Some scheduling strategies require additional parameters (e. g., the execution time).</td>
</tr>
<tr>
<td>TERMINATE &lt;tid&gt;</td>
<td>Termination of task execution.</td>
</tr>
<tr>
<td>SUSPEND &lt;tid&gt;</td>
<td>Temporal suspension of task execution.</td>
</tr>
<tr>
<td>CONTINUE &lt;tid&gt;</td>
<td>Continuation of task execution after the SUSPEND instruction.</td>
</tr>
<tr>
<td>ENTER CRIT</td>
<td>Beginning of the critical section. Scheduling is disabled.</td>
</tr>
<tr>
<td>LEAVE CRIT</td>
<td>End of critical section. Scheduling is resumed.</td>
</tr>
<tr>
<td>LOCK &lt;tid&gt;,&lt;semaid&gt;</td>
<td>Try to lock a synchronizer. If it is not available, a task is suspended.</td>
</tr>
<tr>
<td>UNLOCK &lt;semaid&gt;</td>
<td>Unlock a synchronizer if no task is waiting for it. Otherwise, continue the execution of the first task that is waiting for it.</td>
</tr>
<tr>
<td>TIMESLICE</td>
<td>End of the time slice period for time-sharing scheduling.</td>
</tr>
<tr>
<td>TICK</td>
<td>Periodic update of the internal counters for the schedulability test.</td>
</tr>
</tbody>
</table>

Pseudo instructions

<table>
<thead>
<tr>
<th>Co-processor Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETSV &lt;addr&gt;,&lt;value&gt;</td>
<td>Set a shared variable.</td>
</tr>
<tr>
<td>GETSV &lt;addr&gt;</td>
<td>Read a shared variable.</td>
</tr>
<tr>
<td>RESETWD</td>
<td>Reset watch-dog timer.</td>
</tr>
<tr>
<td>TIMESLICE</td>
<td>End of the time slice for time-sharing.</td>
</tr>
<tr>
<td>SETCR &lt;crno&gt;</td>
<td>Set control register.</td>
</tr>
<tr>
<td>GETSR &lt;srno&gt;</td>
<td>Get status register.</td>
</tr>
<tr>
<td>GETCID</td>
<td>Read an identity number of a task that is currently running on the processor.</td>
</tr>
<tr>
<td>GETDBG &lt;regaddr&gt;</td>
<td>Read one of the internal registers. Used mostly for debugging purpose.</td>
</tr>
</tbody>
</table>