DOI. 10.1109/SACI.2011.5873001

Requirements of a Real-Time Multiprocessor Operating System for Multimedia Applications

Valentin STANGACIU*, Georgiana MACARIU*, Mihai V. MICEA*,
Valentin MURESAN**, Brendan BARRY**
* "Politehnica" University of Timisoara, Timisoara, Romania
** Movidius LTD, Dublin, Ireland
valys@dsplabs.cs.upt.ro, georgiana.macariu@cs.upt.ro, micha@dsplabs.cs.upt.ro,
valentin.muresan@movidius.com, brendan.barry@movidius.com

***Abstract*—Many modern embedded applications need the execution support of a real-time operating system. Such applications have usually critical or hard timing requirements, but giving the high presence of multimedia devices, real-time operating systems can also be used to manage the tasks that execute within multimedia devices. This paper focuses on identifying the requirements of a real-time operating system capable of fulfilling the needs of the multimedia applications typically executed on multiprocessor, mobile-oriented systems such as the Movidius platform.**

I. Introduction

Real-time operating systems are usually used in critical applications, where data acquisition and processing in a strict timely fashion is required, and hard deadlines for system response to events are imposed [1]. In this case not only the result of the task execution is important but also its response time (i.e., the time elapsed until the result is available) [2]. Such real time operating systems [3]-[7] use hard real time algorithms to schedule the tasks that need to be executed within the system [2], [8]-[10].

In multimedia, a class of applications is dedicated to video processing where each frame of a movie can be divided in several slices and each slice can be processed in parallel, speeding-up the processing. Such an example of a real time operating system usage in context with multimedia application is a *"very small on-chip multimedia real-time operating system for embedded system LSIs"* which *"demonstrates its usefulness on MPEG-2 multimedia applications"* [11]. Real time operating systems are also present on DSPs integrated on PC adapter cards with multimedia processing functions [12].

II. The Movidius Platform

The platform used for this approach, designed and produced by Movidius LTD, is a unique and flexible, multi-core architecture (Figure 1) that "*enables a broad range of applications to be developed that previously were not possible on a mobile platform"*. The platform gives 20 GFLOPs of processing power maintaining low power consumption [16].

III. Requirements of a Multimedia-Oriented Real-time Operating System

A generic video processing application will create several threads, each performing a set of operations on a slice of a video frame. Basically, all threads execute the same instructions on different input data (single instruction multiple data parallelism). During video streaming, frames are rendered at a given rate (e.g. 30 frames/second). Therefore, each thread should finish the slice processing before the arrival of the next frame, when it will receive a new slice. In this context, the operating system will be responsible for scheduling the threads on the available cores of a multiprocessor platform such that all threads meet their deadlines. Generally, the RTOS will have to deal with soft deadlines. This means that missing a deadline is not critical. For example, if a frame cannot be processed within a given deadline, the frame can just be dropped while the system displays the previously processed frame. Furthermore, because each thread periodically receives new input data, its execution will resume periodically, and therefore, the operating system should also support such periodic tasks.

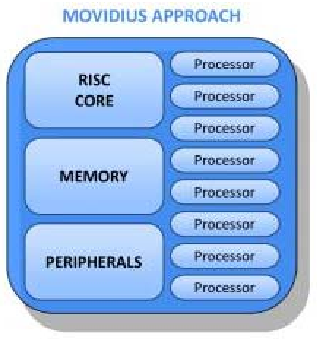

Figure 1. Movidius platform [16]

It is possible that, in order to process a slice of a frame, a thread will require data in slices allocated to other threads. If this data needs to be accessed in mutual exclusive manner, then the operating system will also need to provide thread communication and synchronization mechanisms. However, it is expected that applications will be designed and implemented such that they will require such mechanisms as little as possible.

A system using the Movidius platform should allow for each application to use exclusively a subset of the available cores. The scheduler of the operating system will be responsible for allocating cores to applications and for allocating and relocating application threads to these cores, such that the real-time requirements of each application will be met.

In a real-time operating system determinism and speed are essential and because of these reasons traditional RTOSs have just one memory space, to avoid the overhead and non-determinism (in the case of caching) of address translations. However, when several applications are sharing the same processor platform, if one of them contains faults, it may affect other applications and thus, a protected memory model will be much more secure and error safe.

Furthermore, in order to reduce the errors in applications, the operating system should facilitate application debug and event tracing. Such events will be collected by the RTOS and will include for example thread start time, thread finish time and amount of memory used by a thread.

Based on the considerations above we identify the following functional requirements of a real-time operating system for the Movidius platform:

F1. The RTOS must provide memory protection mechanisms and must provide mechanisms to manipulate the memory management unit (MMU) available on the hardware platform. Further on, translation look-ahead buffers should be used.

F2. The RTOS must implement mechanisms for task communication and synchronization. Since most programming languages for parallel programming (e.g. OpenCL, OpenMP) use barriers as synchronization primitives, the operating system should also support this primitive.

F3. The scheduler of the RTOS must ensure that each thread receives enough processing capacities and reaches its deadlines, whether those deadlines are hard or soft.

F4. The scheduler of the RTOS must schedule threads such that data movement will be minimized.

F5. The RTOS must support periodic execution of threads.

F6. The scheduler of the RTOS must be able to apply a hybrid scheduling policy:

   a. partitioned: each application runs on a specified number of cores,
   b. global: the threads of an application can execute on any of the cores assigned to the application.

F7. The RTOS must implement facilities for application debug and event tracing.

Beside these functional requirements, an RTOS should also meet several non-functional requirements:

N1. The scheduler should operate in a minimum amount of time (e.g. minimum swap time).

N2. The RTOS should include power saving features. For example, if a core is idle, the RTOS may decide to turn off its clock.

N3. The memory footprint of the RTOS must be below 30 KB, to fit in one of the internal memories of the processor.

N4. In the case of existing RTOSs, they should provide support for deploying scaled-down versions of the product, containing only the needed functions.

IV. RTOS Selection

In what follows, an analysis of existing RTOSs is presented. We have selected several RTOSs, commercial and open-source, and analyzed them based on the requirements introduced previously. All selected systems support the SPARC Leon processor, implemented by the operating system owner or as a port provided by Aeroflex Gaisler. The analyzed RTOSs are:

- with commercial license: Nucleus, LynxOS, VxWorks, ThreadX, PikeOS
- open-source: eCos, RTEMS, Linux SnapGear, μCLinux

As mentioned in requirement (F1), the RTOS should provide memory protection and should provide mechanisms for working with the hardware MMU. From the RTOSs above, only a small number of them support such features: Nucleus, LynxOS, VxWorks, Linux SnapGear. In the above list, three of the system are commercial and only one is open-source. Only these systems will be considered next.

The (F2) functional requirement says that the RTOS for the Movidius platform should implement mechanisms for communication and synchronization between execution threads. When several threads need to use the same resource (e.g. data structure, I/O device), that resource must be protected using services in the operating system. Mutexes represent the service providing exclusive access to resources. Mutexes can also be implemented as binary semaphores. Referring back to the last list of existing RTOSs, all of them implement semaphores as synchronization mechanisms, but only LynxOS, VxWorks and SnapGear provide mutex services. For communication between threads all use messages.

The requirement (F3) states that the RTOS should guarantee hard deadlines for at least a subset of the threads. Only Nucleus, VxWorks and LynxOS give this guarantee. All three RTOSs are under a commercial license.

Further, requirement (F4) says that it would be desirable for the chosen RTOS to provide support for working with periodic tasks. None of the operating systems which passed the previous requirements filters provides such services.

The remaining analyzed operating systems (Nucleus, VxWorks, LynxOS) schedule the threads of the applications using a preemptive proprietary policy. The only RTOS that allows processor partitioning between applications in the systems and enables each application to use a different scheduling policy is LynxOS. LynxOs uses para-virtualization to ensure this separation between applications. However, para-virtualization has the drawback of requiring adaptations of the guest applications.

All commercial RTOSs provide services for system debug and profiling. The Linux SnapGear uses gdb for debug purposes, like most Linux implementations.

Furthermore, one of the non-functional requirements of the RTOS specifies that its footprint should be below 30KB. For the analyzed RTOSs the documentation mentions that they have a small footprint, but either does not mention an actual size or there is no information about the conditions under which the mentioned size is achieved.

V. Application Management

From the perspective of the operating system, the applications are composed of one or more processes. A process is an instance of a program in execution and its purpose is to act as an entity to which system resources (CPU time, memory, etc.) are allocated. In the case of multi-threaded applications, which have several execution flows sharing a large amount of application data, a process is composed of several threads, each of which represents an execution flow of the process. Thus, a multi-threaded application is a thread group which shares an address space and other resources.

To manage the application threads, the RTOS must know exactly what each of them is doing. For this purpose, the RTOS will keep the following information about each thread:

- thread ID: an identifier of the thread; this will be set automatically by the RTOS and returned to the creator of the thread
- priority: should be set when the task is created which address space is using
- its current state: during its life cycle, a thread can be in one of the following states:
    - Ready: the thread is waiting for CPU resources,
    - Running: the thread is executing on a CPU,
    - BusyWaiting: the thread required a spin mutex but the mutex is not available; while in this state, the thread will not be preempted from the CPU
    - Blocked: the thread is blocked waiting for a resource other than CPU; this state is reached when the thread wants to acquire a blocking mutex but the mutex is not available; while in this state the thread will yield the CPU and when the mutex granted to it will go to the Ready state, waiting for a CPU to be allocated to it
    - Sleeping: when the sequence of instructions which must be executed periodic by a thread finishes, the thread sleeps until the beginning of the next period; when the new period starts the thread will restart executing the same sequence of instructions
    - Finished: the thread completely finished its work.

The transition between these states is depicted in Figure 2. After the thread is created it will enter the Ready state, waiting for the scheduler to allocate a CPU to it (the “thread created” transition). When a CPU is allocated to the thread, it will start running on that CPU (the “CPU granted” transition). If, during its execution, the thread requests a spin mutex and the mutex is currently locked by another thread, the thread will enter the BusyWait state until the mutex can be granted (the “spin mutex unavailable” transition). While the thread on a core is in the BusyWait state, the RTOS may decide to turn off the clock of the core in order to save power. When the spin mutex can be granted to the thread, the thread will go back to the Running state (the “spin mutex granted” transition). When, during its execution, the thread requests a blocking mutex, if the mutex is currently lock by another thread, the thread will go to the Blocked state yielding the CPU on which it was running (the “blocking mutex unavailable” transition). While the thread is in the Blocked state, the RTOS can use the CPU to execute another thread. When the spin mutex can be granted to the thread, the thread will go to the Ready state, waiting for the scheduler to allocate a CPU to it (the “blocking mutex granted” transition).

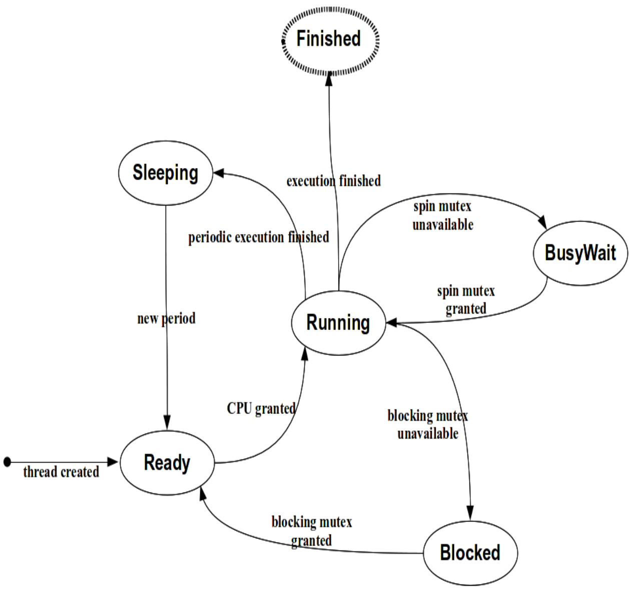

Figure 2. Application thread states

If the thread is periodic, after it finishes the sequence of instructions to be executed in each period, the thread will go to the Sleeping state (the “periodic execution finished” transition). When this state is reached and the thread does not need any resources (e.g. data buffers), all of these should be freed. When a new period starts, the thread will go again to the Ready state (the “new period” transition).

The application and thread information are kept in descriptors. The structure of these descriptors is shown in Figure 3.

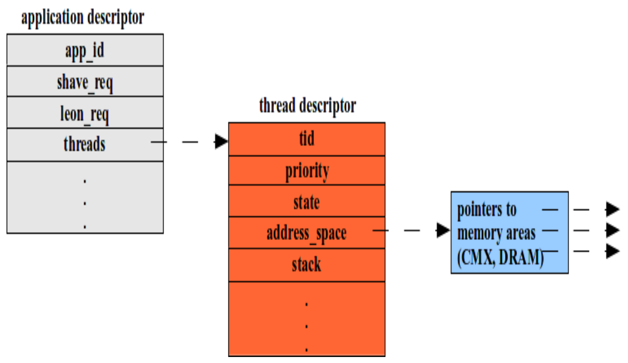

Figure 3. Application and thread descriptors

Each application in the system has its own resource requirements: core time bandwidth, RISC core time bandwidth, etc. Such requirements form the interface (or the contract) of the application. These requirements are kept in fields of the application descriptor and, based on them, the RTOS scheduler will assign processors to it and will ensure that malfunction of any of the applications in the system does not endanger other applications. In the case of the Movidius platform, each application may require a specific scheduling policy (specific priority assignment policy, specific policies for handling delays due to thread synchronization). More details on the scheduling strategy will be presented in the following section.

VI. Supporting Application-Specific Schedulers

Complex embedded software systems are developed from several modules or subsystems, each implementing a well specified function. Each of the subsystems can be developed, tested and validated independently, and later, all the subsystems are integrated in the final system which must also function correctly. As these software systems have real-time requirements, their correct functioning is determined not only by the logical result delivered by the system but also on the time when this result is delivered. When integrating a system from independently developed subsystems, one must ensure that the temporal constraints of each subsystem are met and that malfunctioning of one of them will not influence the correctness of the others.

Therefore, advanced techniques must be employed in order to ensure temporal and spatial isolation of the subsystems, in all development stages, from design until validation. Temporal isolation between real-time subsystems can be ensured using techniques such as hierarchical scheduling introduced in [15]. In hierarchical scheduling, a system is hierarchically composed from several subsystems scheduled by a global system-level scheduler. Each subsystem consists of a set of threads which are scheduled by a local subsystem-level scheduler. Using such a scheduling mechanism, one can analyze each subsystem in isolation, based on a subsystem specific scheduling policy, and later, analyze the integrated system, based on an arbitrary global scheduling policy. Hence, hierarchical scheduling enables concurrent development of subsystems. Furthermore, usage of hierarchical scheduling techniques is appropriate for systems composed of applications with hard-, soft- and non-real-time constraints [14].

In the hierarchical scheduling framework, each subsystem admitted in the system has an interface specifying its minimum resource requirements. The requirements include, but are not limited to, processors, I/O devices or networking devices. Based on this interface, the system automatically defines resource partitions for the subsystem. If we refer to processors, the resource partitions can be viewed as a set of virtual processors which can be mapped either to a physical processor or to a part of a processor. The virtual processors are just mechanisms for ensuring none of the subsystems uses more resources than initially specified.

The mapping of virtual processors to physical processors can be done static or dynamic. In a static setup, before the subsystem starts executing, the virtual processors are created and mapped to physical processors according to the requirements in the interface. For example, in Figure 4 the interface of subsystem 2 (which is an OpenCL driver) could specify that it requires 230% of the processors and 7% of the capacity provided by the RISC core processor. The system can statically define a minimum of 4 virtual processors for this subsystem, as follows: two virtual processors are mapped to two physical processors (any available), one virtual processor can be mapped to a third processor with the restriction that it can use only 30% of it and the forth should be created for the core processor. Although the implementation overhead of this static partitioning is reduced, if there are time intervals when the subsystem does not fully use the allocated processors, they cannot be used by other subsystems and remain idle.

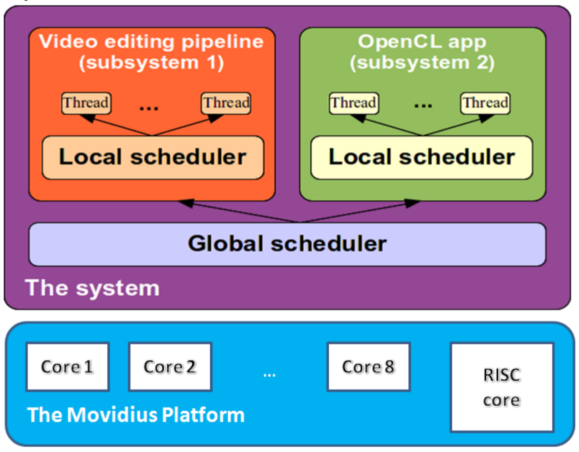

Figure 4. Application specific schedulers

The dynamic mapping removes this drawback. For the example in Figure 4, we can also define a minimum of 4 virtual processors, but they can migrate from one processor to another. We only have to enforce that whenever at least two threads of the subsystem are ready for execution, they will be executing and will not be put on hold due to execution of some other subsystem.

VII. Memory Management

The main requirements placed on the memory management system are:

- minimal data movement,
- minimal fragmentation,
- minimal management overhead, and
- deterministic allocation time.

In the case of the Movidius platform, the RTOS must manage the CMX and the DDRAM memory. For the DDRAM memory, in order to achieve minimal memory fragmentation and deterministic allocation time the RTOS will provide memory pools composed of fixed-size memory blocks. Depending on its characteristics, each application will create memory pools for the required DDRAM memory and all next memory allocation demands will be served from that pool. Thus, memory pools allow memory allocation with constant execution time and reduce the memory leaks within one application the programmer has to release only the memory pool when the application finishes and not each requested block.

For example, in Figure 5 the memory space is divided in two memory pools with blocks of 32 bytes and 128 bytes, respectively (the values were chosen arbitrarily). Each memory pool is characterized by its block size, number of blocks in the pool, number of free blocks, and linked list of free blocks. If one application requests 64 bytes of memory from pool A then the first two blocks from the pool will be returned.

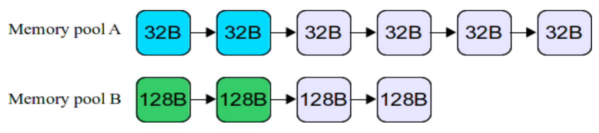

Figure 5. Fixed-size memory pools

For the CMX memory, the RTOS will allocate memory in the CMX slice of the processor on which the thread that requested it is running. By this, the RTOS will try to minimize data movement. In the same sense, if the thread to be scheduled needs data produced by a previously scheduled thread, the scheduler of the RTOS will try to start the new thread as closer to the data as possible. If the requested memory block cannot be allocated in the processor slice, the RTOS will search for another slice such that the number of potential stalls is minimized. For the CMX, the RTOS may use the SLOB memory allocation algorithm.

VIII. USAGE SCENARIOS

This section presents how two of the applications running on the Movidius platform may use the above described functionality of the RTOS.

A. Video Editing Pipeline

The Video Editing Pipeline (VEP) application receives as input a video (from the camera or from an existing file) and applies a set of effects (e.g. overlay, old movie, cartoon, etc) on each frame of the video. The effects are applied sequentially, meaning that an effect is applied only after the previously one is applied to all frames of the video.

If we assume that frames arrive at a rate of 25 frames per second, then applying a certain effect on a video can be implemented by a periodic thread with a period of 40 msec. and every time it receives a frame it must finish processing it before the next frame arrives. In order to speed-up the video processing, each frame is divided in several slices and a thread is created for applying an effect on each slice. Based on the rate of the frames and on the average time required to process the slice, the minimum and maximum number of slices and threads can be determined. These represent the minimum and maximum number of processors required by the application. This data will be used by the scheduler of the RTOS to allocate processing resources to the application. The threads will then be scheduled non-preemptively on the processors.

The frames are loaded in DDRAM memory and from there, the input and output slices for a thread are buffered in the CMX memory slice of the core on which the thread is running or in the slice which will cause minimum number of clashes (see Figure 6). Therefore the application can create a memory pool for loading the frames in the DDRAM memory.

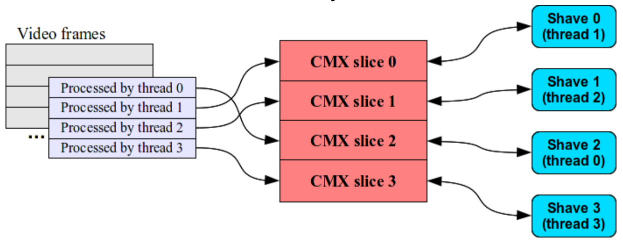

Figure 6. Video editing pipeline

B. Executing an OpenCL application

An OpenCL application is the combination of a program running on the host and OpenCL devices. For the Movidius platform, the host is the RISC core while the devices are the rest of the cores. Execution of an OpenCL application occurs in two parts: kernels that are executed by the processors and the host program executed by the core processor. The host program creates the kernels and manages their execution by creating command queues associated with each processor. Thus, in this case, the host program is responsible for the actual scheduling of the kernels and the hierarchical scheduling framework provides the support for implementing such application specific scheduling strategy.

The memory model of OpenCL, depicted in Figure 7, can easily be mapped to the architecture of the Movidius platform as the CMX memory slices are the Local memory for each processor and the DDRAM memory is the Global memory.

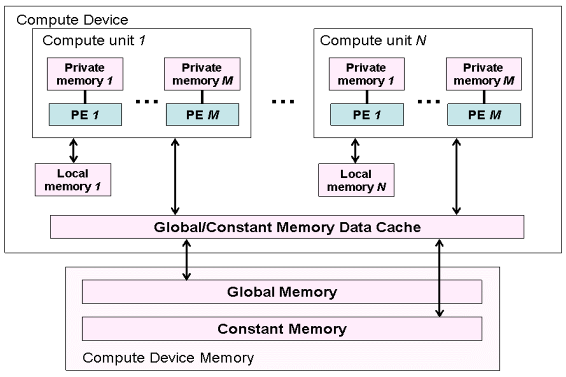

Figure 7. OpenCL memory model [13]

IX. WRITING PARALLEL APPLICATIONS

This section describes the data structures and the application programming interface (API) for writing parallel applications to be executed on the Movidius platform.

A. Capturing data parallelism

To be able to break the data collection on which the parallel processing will be applied, the API will contain types for defining iterators. Such types must support operations for:

1. testing if the collection is empty,
2. specifying how the data collection should be split
3. retrieving data elements from the iterator.

The API will provide three operations or functions for enabling data parallelism:

1. *parallel_for*: breaks a data collection into chunks and processes each chunk in a separate thread (the data collection must have an associated iterator); the threads will be created transparently and will not be the task of the developer although the developer may have the possibility to specify a minimum degree of parallelism.
2. *parallel_foreach*: processes each data item in a collection in a separate thread the threads will be

created transparently and will not be the task of the developer.

3. *parallel_reduce*: computes reduction (e.g. computes a sum) over chunks of data using several parallel threads; the threads will be created transparently and will not be the task of the developer although the developer may have the possibility to specify a minimum degree of parallelism.

For example, the *parallel_for()* operation can be used for the H.264 decoder in order to process in parallel several macro-blocks. The *parallel_reduce()* operation can be used in image processing application which may require performing sums on pixels on each column/row.

B. *Capturing task parallelism*

Task parallelism refers to one or more independent tasks running concurrently. A task represents an asynchronous operation and will ultimately result in the creation of a new thread. The API will provide the possibility to define tasks as well as to manage their execution:

- wait for its finish,
- cancel it,
- check its status,
- check its result
- define task precedence constraints either by specifying the tasks on which a task depends or by specifying the tasks which depend on a certain task.

The API will provide the following operations or functions for enabling task parallelism:

- *parallel_invoke(list of functions)*: given a list of functions, a separate task will be created for each of them and all tasks will be scheduled in parallel;
- *task_start*: creates a task and executes it asynchronously,
- *task_start(list_of_tasks)*: creates a task and specifies that its execution must not be started until the tasks in the list of tasks given as parameter finish,
- *task_stop*: cancels a task; all tasks depending on it will also be canceled
- *task_result*: get the result of a task
- *task_status*: get the current status of the status (status refers to: started, waiting for tasks on which it depends to finish their execution, finished, canceled); also if the code executed by the task is annotated and checkpoints are defined for it, the operation will return the last checkpoint reached during execution.

C. *Mutual exclusion*

The API will also provide support for mutual exclusion by software mutexes. For implementing the software mutexes the RTOS will reserve one of the hardware mutexes. The other hardware mutexes can be used by the applications. The API will define at least two types of mutexes:

1. *a spin mutex*: a thread trying to acquire such a mutex busy waits until it acquires it; this mutex is suitable for short critical sections.
2. *a blocking mutex*: a thread trying to acquire such a mutex will yield the processor and will block until the mutex will be granted to it; this mutex is suitable for long critical sections.

For each kind of mutex the API will provide operations for acquiring and releasing the mutex:

- *lock(mutex)*: lock the given mutex,
- *unlock(mutex)*: unlock the given mutex.

D. *Other synchronization mechanisms*

Beside mutexes, the API defines also other synchronizations methods. In a first iteration, it is expected that the API will provide support for:

- *events:* an event can be used to track the execution of a thread and thus, to notify one or several threads that the thread producing the event has reached a certain point in its execution.
- *barriers:* this mechanism is used for blocking its user - the thread that called it - until all threads that also use the barrier reach it.

X. CONCLUSIONS

This paper focuses on identifying the requirements of a real-time operating system (RTOS) that is meant to be the manager of the Movidius platform, a multi-core processor designed for multimedia applications for the next generation of mobile devices. The real time operating system derived from the requirements discussed in this paper is a soft real-time, deadline oriented, RTOS, mainly used for video processing applications.

REFERENCES

[1] M.V. Micea, C.S. Certejan, V. Stângaciu, R. Cioargă, V. Creţu, E. Petriu, "*Inter-Task Communication and Synchronization in the Hard Real-Time Compact Kernel HARETICK*", Proc. IEEE Intl. Wshop. Robotic and Sensors Environments, ROSE 2008, Ottawa, Canada, pp. (18-24), 2008.

[2] G. Butazzo, P. Gai, "*Efficient EDF Implementation for Small Embedded Systems*", Proc. Intl. Wshop. Oper. Syst. Platforms for Embed. Real-Time Applic., OSPERT 2006, Dresden, Germany, 2006.

[3] M. V. Micea, *"HARETICK: A Real-Time Compact Kernel for Critical Applications on Embedded Platforms"*, Proc. 7th Intl. Conf. Development and Applic. Syst., DAS2004, Suceava, Romania, pp. (16–23), May 2004.

[4] OSEK Group, "*OSEX/VDX Operating System Specification*", Version 2.2.3., Feb. 2005, [Online: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf].

[5] I. Ripoll, P. Pisa, L. Abeni, P. Gai, A. Lanusse, "*RTOS State of the Art Analysis*", Deliverable D1.1 – RTOS Analysis, OCERA 2002.

[6] VxWorks, "*VxWorks Programmer's Guide*", Rev. 5.5. Wind River Systems, Inc., Alameda, CA (SUA), 2002.

[7] K. Cheung, "*OSE Real-Time Operating System*", Embedded Star, Jul. 2006, [Online: http://www.embeddedstar.com/weblog/2006/07/07/ose-real-time-operating-system/].

[8] M.V. Micea, V. Cretu, *"Highly Predictable Execution Support for Critical Applications with HARETICK Kernel"*, Intl. J. Electronics Communic. (AEUE), 5 (59), Elsevier, pp. (278−287), 2005.

[9] L. George, N. Rivierre, M. Spuri, "*Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling*", Raport de Recherche, Institut National de Recherche en Informatique et en Automatique (INRIA), 1996.

[10] P.G. Jansen, S.J. Mullender, P.J.M. Havinga, H. Scholten "*Lightweight EDF Scheduling with Deadline Inheritance*", Internal Report, University of Twente, 2003.

[11] H. Iwasaki, J. Naganuma, M. Endo, T. Ogura, "*On-Chip Multimedia Real-Time OS and Its MPEG-2 Applications*", Proc. 6th Intl. Conf. Real-Time Comput. Syst. and Applic., RTCSA 1999, Hong Kong, China, pp. (200-203), 1999.

[12] D. Katcher, K. Kettler, J. Strosnider, "*Real-Time Operating Systems for Multimedia Processing*", Proc. Wshop. Hot Topics in Oper. Syst., pp. 18, 1995.

[13] Khronos OpenCL Working Group, A. Munshi, *"The OpenCL Specification"*, Version 1.0, Rev. 48, Khronos Group, 2009.

[14] B.B. Brandenburg, J.H. Anderson, "*Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors*", Proc. 19th Euromicro Conf. Real-Time Syst., ECRTS 2007, pp. (61−70), 2007.

[15] C.W. Mercer, S. Savage, H. Tokuda, *"Processor Capacity Reserves for Multimedia Operating Systems*", Tech. Rep., Pittsburgh, PA, USA, 1993.

[16] ***, http://www.movidius.com