Grant Agreement number: 288574

Project acronym: **vIrtical**

Project title: SW/HW extensions for virtualized heterogeneous multicore platforms

Seventh Framework Programme

Funding Scheme: Collaborative project

FP7 -ICT -2011-7

Objective ICT-2011.3.4 Computing Systems

Start date of project: 15/07/2011 Duration: 36 months

D 3.3 *Hypervisor for distributed time- and event-triggered threads*

Due date of deliverable: M18

Actual submission date: 08.02.2013

Organization name of lead beneficiary and contributors for this deliverable: SYSGO,VOSYS, Thales

Work package contributing to the Deliverable: WP3, Task 3.2

Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	X
Explanation of dissemination level change from PU to CO: The document provides advanced technical details about the behaviour and techniques used in the hypervisors to provide the required capabilities, and since some of these techniques are still in development, then it is reasonable not to have them available at PU level.		

APPROVED BY:

Partners	Date
All	6th March 2013

INDEX

1 Abstract 4
Glossary 5
2 Introduction 6
3 Scheduling 6
3.1 Time-triggered scheduling 6
3.2 Event-triggered scheduling 7
3.3 Heterogeneous multi-core scheduling 8
3.3.1 Big.LITTLE scheduling configurations 8
3.3.1.1 Migration model 9
3.3.1.2 Cluster Migration 9
3.3.1.3 CPU Migration 10
4 Mixing time-triggered and event-triggered scheduling on MP systems 11
4.1 Requirements 11
4.1.1 Functional Requirements 12
4.1.2 Quality Requirements 12
4.1.3 Non-functional Requirement 12
4.2 Scheduling Algorithms 12
4.2.1 Approach 1: Time-partitioning with background partition 13
4.2.1.1 Background time partition 13
4.2.1.2 Time-partitioning multi-core with SMP 15
4.2.1.3 Scheduling on multi-core systems with time partitioning 15
4.2.1.4 Implementation on big.LITTLE 16
4.2.1.4.1 Cluster migration 16
4.2.1.4.2 Cluster power-up/down 17
4.2.2 Approach 2: Extending the CFS scheduler for heterogeneous platforms ...17
4.2.2.1 CFS scheduler limitations 17
4.2.2.2 big.LITTLE kernel scheduling 18
4.2.2.3 Power efficient scheduling and monitoring metrics 18
5 Conclusion 19

1 Abstract

In this report we consider event- and time-triggered scheduling algorithms, which target safe/secure and efficient usage of the heterogeneous ARM big.LITTLE platform. We present two different approaches on how to exploit this new functionality provided by the hardware platform. First approach focuses on safety and security critical tasks in the presence of time and event-triggered tasks. The second approach is based on Linux/KVM with focus on optimising power consumption.

Glossary

Name	Description

2 Introduction

In this document we consider time-triggered and event-triggered scheduling algorithms on heterogeneous multi-core systems. These algorithms are either part of the hypervisor (virtual machine monitor) or in the case of KVM, part of the host kernel itself, i.e. they schedule threads/tasks from different virtual machines (VM). Schedulers inside a VM (user level schedulers) are out of scope of this document.

The reference hardware platform for this document is the ARM big.LITTLE architecture. big.LITTLE consists of two architecturally equal CPU clusters - both using the ARMv7 architecture. The "big" CPU cluster includes up to four high performance Cortex-A15 cores. The "LITTLE" CPU cluster includes also up to four less performance but energy efficient Cortex-A7 cores. The focus of this document lies on how to configure and use the reference platform to achieve efficient co-existing of time- and event-triggered threads.

It is important to define the scope of this document in the vIrtical project architecture. We focus exclusively on hypervisors. Other architectural blocks considered in the project (GPPA, NoC) could also support the distribution of event and time triggered threads. For example, the STM Spidergon NoC provides support for quality of service (QoS), the internal scheduler of GPPA can assist hypervisor/OS to more efficiently achieve distributions. These hardware functionalities are not part of this deliverable and are treated in other deliverables of WP3 and WP4.

3 Scheduling

In this section we briefly describe two considered approaches for scheduling: time-triggered and event triggered scheduling.

3.1 Time-triggered scheduling

Time triggered scheduling means that all relevant events are processed at predefined points. Usually time-triggered scheduling is defined before the system is deployed, i.e. the scheduler configuration is made offline. Such a scheduler is executed periodically, which guarantees predictable and repetitive scheduling of system processes.

The main part of such a scheduler is the configuration. In a configuration one splits some amount of time (e.g. 1 second) into *slots* (also known as time windows, time frames, etc) and assigns processes/threads to be active at specific slots. The result of the splitting and assignment is one period of the time-triggered scheduling. In this document we call this period *major time frame*. The major time frame is repeatedly executed all over again during the whole lifetime of a system.
During runtime the scheduler makes lookups in the configuration to decide if the next slot has to be activated.

Time-triggered scheduling has the following characteristics:

- Complexity is moved to offline configuration allowing to keep implementation simple and runtime system small
- Smallest slot in major time frame defines the time granularity
- Reaction time is, in the worst case, the length of the major time frame

Time-triggered scheduling has the following advantages:

- Simpler and faster dispatcher/scheduler design
- Easier certification of implementation due to the simpler design
- Simple fault tolerance, easier introduction of soft deadlines and health monitoring of time consumption
- Deterministic execution, you get what you planned
- Simpler testing

- Offline (i.e. before deployment) analysis of complex scheduling, tool support can be used (scheduler planners)
- Easy integration of non-temporal requirements, e.g. energy consumption
- High-resource utilization, i.e. less over-provisioning due to planning

Time-triggered scheduling has the following disadvantages:

- Need to know everything from the beginning, including
  - all external signals and time of their occurrence/arrival
  - all system parameters for the system lifetime
- Not flexible, e.g. creating new tasks with new scheduling requirements
- Periodic execution, e.g. fixed periods
- Not-used time is wasted, i.e. over-provisioning cannot be utilized

3.2 Event-triggered scheduling

In the event-triggered scheduling, the execution of processes is driven by the occurrence of some events that are related to a change in the system. Usually, the system functionality is composed of several processes or virtual machines and their execution might lead to resource conflicts, such as the case when two processes are simultaneously ready for execution and only one of them can make use of the processor capabilities of the system. A way to resolve such conflicts is by assigning priorities to processes. In this scenario the processes with highest priority are executed at the beginning.

One of the simplest and most common scheduling is called the fixed priority scheduling, in which the priorities are statically assigned to processes offline. In order to implement a fixed priority scheduling algorithm an OS includes a scheduler which has two main responsibilities:

- Maintain/update the prioritized queue of ready tasks.
- Select from the queue and execute the ready task with the highest priority.

Depending on the possibility to de-schedule a process when processes with highest priorities are available in the ready queue, we can say that the OS implements a pre-emptive policy. In this case, the de-scheduled process is placed in the ready queue and it will be resumed after a process with higher priority has finished its execution. In the opposite case, called non-pre-emptive, the higher priority process has to wait until the process currently executed has finished its execution. This means that processes with higher priority can be blocked for an amount of time in the ready queue until a subsequent activation of the OS scheduler. The main advantages of the event-triggered scheduling are its flexibility and an efficient usage of the available resources.

Linux kernel is a multi-module system, including process scheduling, memory management, communication between processes, virtual file system and network interface, KVM etc. Meanwhile, process and virtual machine scheduling is a key function in an OS, and a complicate operation which requires several system modules to finish together. The scheduling algorithm of traditional Linux must fulfil several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low/high priority processes, and so on. The set of rules used to determine when and how selecting a new process to run, is called scheduling policy.

Linux scheduling is based on the time-sharing technique: several processes are allowed to run concurrently, which means that the CPU time is roughly divided into slices, one for each runnable process. Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or quantum expires, a process switch may take place. Time-sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs in order to ensure CPU time-sharing.

The Linux OS supports default time-sharing policy and real-time scheduling policy, which includes FIFO and Round-robin (RR) policies defined by POSIX. FIFO and RR policies used to control real-time tasks are introduced in detail:

1. A First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority process is runnable, the process will continue to use the CPU as long as it wishes, even if other real-time processes having the same priority are runnable.
2. A Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all SCHED_RR[1] processes that have the same priority.

Process scheduling in Linux mainly depends on two functions: schedule() and scheduler tick(). The former one is used to select a new process and implement context switch. It is invoked, directly or in a lazy way, by several kernel routines. Its objective is to find a process in the runqueue list and then assign the CPU to it. And the latter is employed to update time slice and check whether the process needs to be pre-empted.

3.3 Heterogeneous multi-core scheduling

Traditionally, it has not been possible to design a processor that offers high performance and high energy efficiency. Solutions have typically involved integrating processors with microarchitectures optimized for performance and energy efficiency respectively. An example is a high performance application processor coupled with a low power ASIC.
This concept, heterogeneous multiprocessing, involves putting a number of specialised cores together – such as, say an application processor and a baseband radio processor – and running tasks suited to each specialized core as needed.

ARM's solution is to couple a high performance processor with a highly energy efficient counterpart while retaining full instruction set architecture compatibility. This coupled design, known as big.LITTLE processing, provides new options for matching compute capacity to workloads by enabling optimal distribution of the load between the 'big' and 'LITTLE' cores.
The current generation of big.LITTLE design pairs a Cortex-A15 processor cluster with an energy efficient Cortex-A7 processor cluster. These processors are 100% architecturally compatible and have the same functionality – support for LPAE, virtualization extensions and functional units such as NEON and VFP – allowing software applications compiled for one processor type to run on the other without modification.

Both processor clusters are fully cache coherent, enabled by ARM's CoreLink Cache Coherent Interconnect. This also enables I/O coherency with other components, such as a Mali-T604 GPU. CPUs in both clusters can signal each other via a shared interrupt controller, such as the CoreLink GIC-400.

3.3.1 Big.LITTLE scheduling configurations

Since the same application can run on a Cortex-A7 or a Cortex-A15 without modification, this opens up the possibility of mapping applications to the right processor on an opportunistic basis. This is the basis for a number of execution models, namely:

- big.LITTLE migration
- big.LITTLE multiprocessing

Migration models can be further divided into types:

- Cluster migration
- CPU migration

---

[1] SCHED_RR is Linux terminology for processes with round robin scheduling

Migration models enable the capture and restoration of software context from one processor type to another. The software stack only runs on one cluster in the case of cluster migration. In CPU migration, each CPU in a cluster is paired with its counterpart in the other cluster and the software context is migrated opportunistically between clusters on a per-CPU basis.
The multiprocessing model enables the software stack to be distributed across processors in both clusters and all CPUs may be in operation simultaneously.

3.3.1.1 Migration model

Migration models are a natural extension to power performance management techniques such as dynamic voltage and frequency scaling (dvfs). A migration action is akin to a dvfs operating point transition. Operating points on a processor's dvfs curve will be traversed in response to load variations. When the current processor (or cluster) has attained the highest operating point and the software stack requires more performance, a processor (or cluster) migration action is effected (see Figure 1). Execution then continues on the other processor (or cluster) with the operating points on this processor (or cluster) being traversed. When performance isn't needed, execution can revert back.

Coherency is a critical enabler in achieving fast migration time as it allows the state that has been saved on the outbound processor to be snooped and restored on the inbound processor, rather than going via main memory. Additionally, because the L2 cache of the outbound processor is coherent, it can remain powered up after a task migration to improve the cache warming time of the inbound processor through snooping of data values. However, since the L2 cache of the outbound processor cannot be allocated, it will eventually need to be cleaned and powered off to save leakage power.

3.3.1.2 Cluster Migration

Only one cluster, either big or LITTLE, is active at any one time, except very briefly during a cluster context switch. The aim is to stay resident on the energy efficient Cortex-A7 cluster, while using the Cortex-A15 cluster opportunistically.

If the load warrants a change from big to LITTLE, or vice versa, the system synchronises all cores and then transfers all software context to the other cluster. As part of this process, the operating system on every processor has to save its state whilst still operating on the old (outbound) cluster. When the new (inbound) cluster boots, the operating systems need to restore their state on each processor. Once the inbound cluster has restored context, the outbound cluster is switched off.

The mode works most efficiently with a symmetric big.LITTLE system – the same number of cores in both clusters. An asymmetric system would require additional operating system involvement to scale execution down to the least common number of cores before migration could take place. While this is possible, it will increase the latency.

Cluster migration can be implemented alongside existing operating system power management functionality (such as idle management) with about the same complexity.

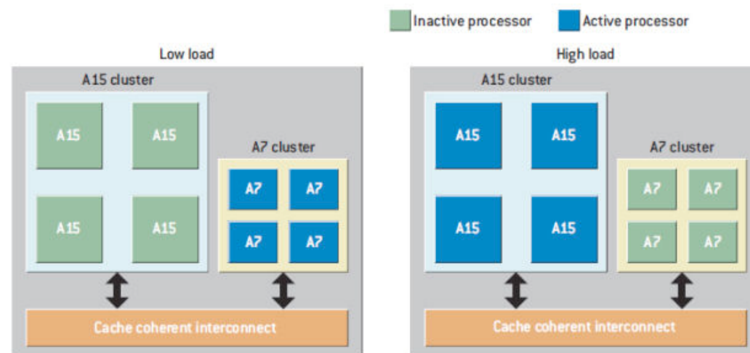

Figure 1: The cluster coherent model

3.3.1.3 CPU Migration

In CPU migration (see Figure 2), each processor on the LITTLE cluster is paired with a processor on the big cluster. CPUs are divided in pairs (CPU0 on the Cortex-A15 and Cortex-A7 processors, CPU1 on the Cortex-A15 and Cortex-A7 processors and so on). When using CPU migration only one CPU per processor pair can be used at the same time.

The system actively monitors the load on each processor. High load causes the execution context to be moved to the big core, while if the load is low, execution is moved to the LITTLE core. When the load is moved from an outbound core to an inbound core, the former is switched off. This model allows a mix of big and LITTLE cores to be active at any one time.

Since a big.LITTLE system is fully coherent through the CCI-400, another model is to allow both Cortex-A15 and Cortex-A7 processors to be powered on and executing code simultaneously. This is called big.LITTLE MP – essentially heterogeneous multiprocessing. This is the most sophisticated and flexible mode for a big.LITTLE system, involving scaling a single execution environment across both clusters. In this use model, a Cortex-A15 processor core is powered on and executing simultaneously with a Cortex-A7 processor core if there are threads that need such a level of processing performance. If not, only the Cortex-A7 processor needs to be powered on. Since processor cores are not matched explicitly, asymmetric topologies are simpler to support with MP operation.

With big.LITTLE MP, the operating system requires a higher degree of customization to extract maximum benefit from the design. For example, the scheduler subsystem will need to be aware of the power-performance capabilities of the different processors and will need to map tasks to suitable processors. Here, the operating system runs on all processors in all clusters which may be operating, migrating tasks between processors in the two clusters simultaneously. Since the scheduler is involved in the directed placement of suitable tasks on suitable processors, this is a complex mode. The OS attempts to map tasks to processors that are best suited to running those tasks and will power off unused, or underused, processors.

Multiprocessor support in the Linux kernel assumes that all processors have identical performance capabilities and therefore a task can be allocated to any available processor. This means support for full big.LITTLE MP requires significant changes to the scheduling and power management components of Linux.

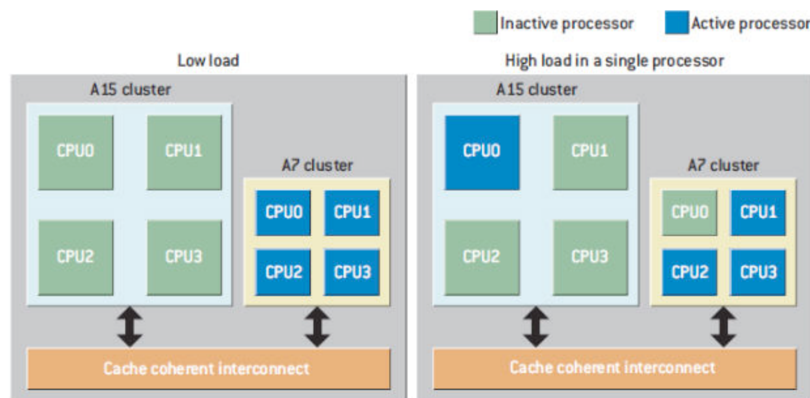

Figure 2: The CPU migration model

4 Mixing time-triggered and event-triggered scheduling on MP systems

Mixing time-triggered and event-triggered threads on MP is a system level task. The hypervisors shall provide sufficient support with required characteristics.

On a uniprocessor/unicore system, there are not many possibilities to mix time and event-triggered threads since:

1. if an event occurs whenever a time-triggered thread is active, we have to wait until the time partition allocated to the thread is over before handling the event. We do not know any way to preempt the time-triggered thread, since this would break the thread's timeliness.
2. if a time-triggered thread gets schedulable whenever an event-triggered thread is running, then we have to preempt the latter and start the former (again for timeliness purposes).

However, on a multi-core system we can do many other things, such as:

- In the first case (see above) we can schedule the event-triggered thread on another core, if there is no other time-triggered thread running on that core.
- In the second case (see above) we can schedule the time-triggered thread on another core, to avoid unneeded preemption, again if possible.

Since events are by nature unpredictable in time, which is not the case of time slots, one way of "organizing" the mix could be to group events on the same core and times on others. Thus, events could compete among one another "randomly" while times could be organized to share the CPU time predictably. However, in this case it is not always so simple because time partitions may overlap one another, which is not possible on uniprocessors. Thus, in such case we might end up in a quasi-static scheduling over a pool of CPUs, which results in organized migrations to fulfill the overlapping scheduling requirements.

Therefore, we identify a set of functional requirements for hypervisors running, to allow an efficient combination of time and event triggered tasks to achieve the expected behavior. We also define qualification requirements (how functionality should be implemented) as well as provide two non-functional requirements.

4.1 Requirements

For efficient combination of time- and event-triggered approaches the following criteria has to be considered:

4.1.1 Functional Requirements

- **Events prioritization**
  In case of two events occurring simultaneously on the system, a priority scheme is needed in order to serialize deterministically the events dispatched by the system.

- **Affinity**
  Threads may be activated only on a subset of the cores available for computations.

- **Dynamic migration**
  A thread can be prevented/enabled to migrate from one core to another at any time, by the system software layer. For critical threads it should be possible to restrict only when the thread is not running (e.g. inactive, waiting, suspended).

- **Common cache control**
  Caches common to several cores (e.g. L2) can be set to unused for all cores except one of them by the system software layer.

- **Mutual exclusion**
  A single thread of the system may be entitled to be runnable while mutually excluding all other threads on all other cores.

4.1.2 Quality Requirements

- **Determinism**
  Whenever a time-triggered thread has to be active, its time slot including its VM has to be active. Thus, the hypervisor shall align scheduling threads from virtual machines with the predefined time-triggered scheduling.

- **Responsiveness**
  The external events, which have to be processed in real-time shall be processed according two exclusive schemes:
  1. with shortest possible delay, for performance-driven mode
  2. with specific bounded delay for security-driven mode. Min and max delays can be provided at system static configuration.

4.1.3 Non-functional Requirement

- **Re-allocation of excess computing time**
  Efficient utilisation of dynamically available computing resources

- **Auditing**
  If needed, events occurrence can be monitored either locally or remotely. It can be done by a facility similar to Linux kernel log in a rotating memory buffer of fixed size.

4.2 Scheduling Algorithms

In this section we describe two algorithms to approach mixing time and event triggered threads.

The first algorithm “Time partitioning with background partition” is implemented in PikeOS and has been extended for vIrtical purposes to account big.LITTLE architecture.

The second algorithm is implemented in the Linux OS, on top of the already available CFS scheduler, extending it for heterogeneous platforms like the big.LITTLE architecture.

4.2.1 Approach 1: Time-partitioning with background partition

Time Partitioning [Kaiser07] shall ensure that the activities in one time partition do not affect the timing of the activities in other time partitions. Time Partitioning shall maintain a major time frame of fixed duration, which is periodically repeated throughout the runtime operation. Time partitions are activated by allocating one or more time-slots within the major time frame. The order of the partition activation is defined off-line, i.e. in static configuration. Time Partitioning should be able

- Allocate a certain amount of CPU time to each virtual machine.
- One time partition can consist of more than one time slot.
- Multiple virtual machines can belong to the same time partition.
- Privileged processes are allowed to create and move threads to time partitions different from the time partition configured for their virtual machine.
- Independent scheduling inside every time partition.
- Support for different priorities inside every time partition.

Time partitioning is very important in the context of mixing safety, security, and real-time requirements. The best example is a safety critical application, which has to be able to work without interference (in this case the corresponding safety requirement “do the job”) and at the same time it should not be possible to exploit scheduling algorithms as a timing covert channel. Time partitioning can also help system integrators to setup a safe access (i.e. access for safety-critical applications) or safe scheduling of accesses.

4.2.1.1 Background time partition

When all threads in a given VM and/or a given time slot have finished all their jobs, in a purely time-triggered partitioning the rest of the time has to be “burned”. If a hypervisor forces switch of time slots, it can destroy the planned scheduling, e.g. next VMs will be scheduled earlier and could finish earlier than the expected arrival of some events.

We suggest the usage of a *background time partition* (we also called it *tau0*) to extend the time-triggered nature of time partitioning with event-triggered and priority-based features.

In a scheduler with a background partition there are always two active time partitions: the current one (according to the static time partitioning) and the background one. The next task is the one with the highest priority from these two partitions. If priority of tasks is equal, then the background partition has precedence. Figure 3 depicts the scheduling decision in PikeOS.

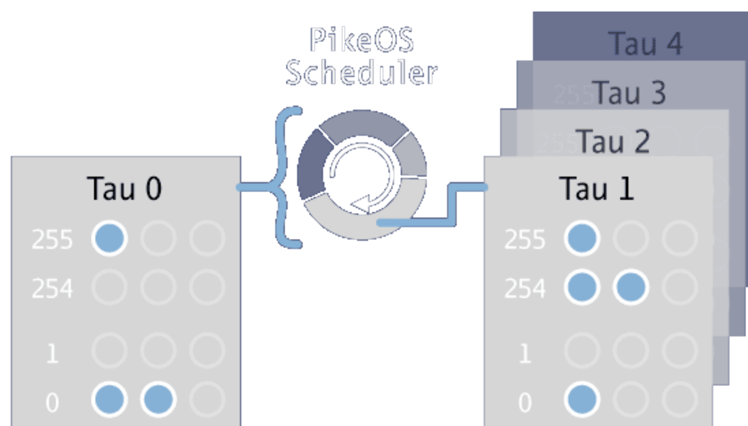

Figure 3: Scheduling with background time partition

This extension allows system integrator to implement the following scenarios:

- Utilisation of free time
  System integrator assigns to the background partition a VM (or threads from a VM) with low priorities. In such configuration if there are not any active threads in current time partition, the threads from background partition will be automatically scheduled. Thus, the available resources will be utilised. After the time slot switch, the priorities of threads from the background partition will be compared against new current time partition. Figure 4 shows advantages of using background time partition with respect to standard time-triggered scheduling defined in avionics standard ARINC 653.

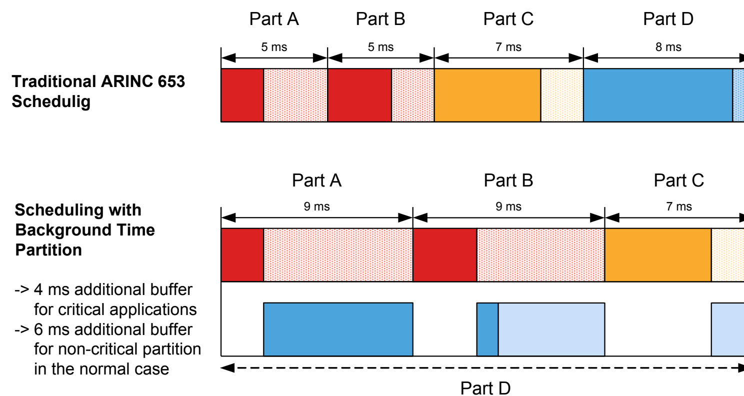

Figure 4: Utilisation of free time through background time partition

- Processing real-time and/or critical events
  System integrators assign to the background partition a VM (or threads from a VM) with a thread which should process some real-time event (or event with the shortest possible reaction time). In a default configuration this thread has the lowest priority, and thus, excluded from scheduling. Once a high-priority events (which is implemented as interrupt) arrives, the interrupt processing routine should rise the priority of the corresponding thread. For example, if it sets the priority of the thread in background partition to the maximum, it will be the next one to schedule. Figure 5 illustrates the described use-case where power fail is the high-priority signal.

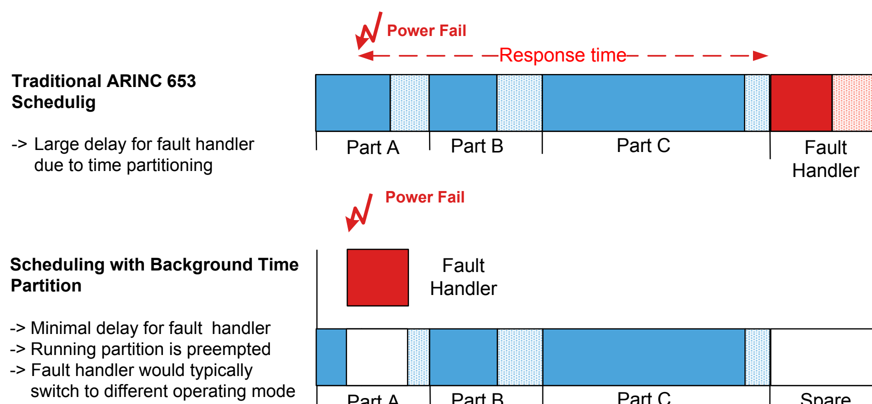

Figure 5: Usage of background partition to achieve shortes reaction time

4.2.1.2 Time-partitioning multi-core with SMP

Time Partitioning on a SMP system shall work the same way as single core processors with independent time partition schemes per core:

- Time partitioning on one processor shall be independent from the time partitioning on another processor.
- Each processor has its own background time partition (tau0) and its own time partition current (tau-Current).
- A time partition is always assigned to a single processor (time partitions never migrate).
- Time partitions are numbered with their logical IDs (tau0 .. tauN) on each processor.
- A time partition scheme shall be defined only for a single processor.
- Each processor has its own time partition switching scheme.
- Time partition schemes shall be configured independently from each other.
- Time partitions with same logical ID on different processors define a Time Domain.
- A domain spans across all processors.

For certain use cases, time partition switching must be synchronized between different processors of the system. This can be achieved easily by using a single time source for time partition switching on all processors. For synchronization of time partition schemes on different processors, switching of one scheme can be synchronized to the start of the major time frame on a different processor.

4.2.1.3 Scheduling on multi-core systems with time partitioning

Threads are the scheduled entities of SMP operating system. The SMP scheduling is realized in three dimensions:

- The first dimension shall be the priority one. The higher the priority, the higher the chance to become CURRENT.
- The second dimension shall be the concept of time-partitioning. Instead of maintaining one ready-queue per priority, a separate ready-queue for each time-partition exists. The rules of the first dimension kept its validity and these time-partitions are scheduled in time. Up to two time partitions can be active at the same time (Tau0 and TauCurrent).
- The third dimension shall be the processor the time partitions are executing on. On each processor, up to two time partitions can be active at the same time (Tau0 and TauCurrent). Time partitions with the same ID make up a time domain.
- Each processor has its own idle thread. At least one thread must be schedulable on each processor.
- Scheduling on one processor does not affect scheduling on another processor, as long as entities of different processors do not interact with each other.
- A thread shall migrate between all three axes on user request.
- Automatic migration between processors is restricted to a special use case discussed below.

Figure 6 shows an example configuration of a system with four virtual machines resources running on a system with four CPUs. There are three time partitions where some of the time partitions are shared between several virtual machines.

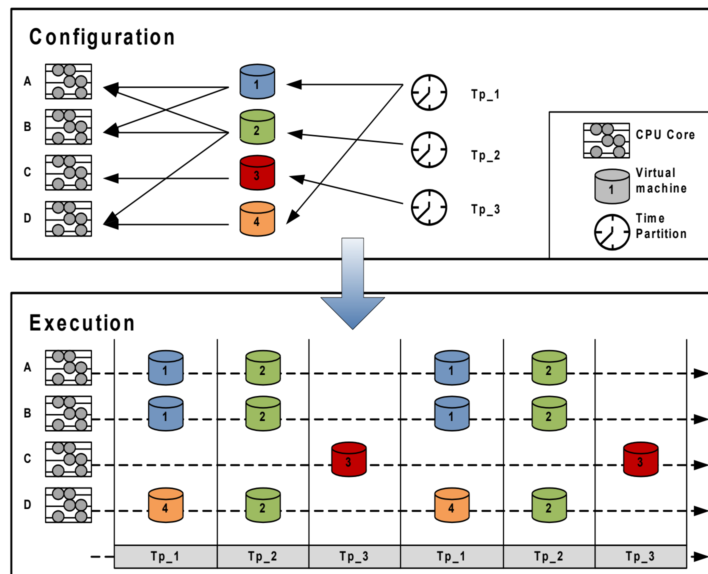

Figure 6: Running four virtual machines on a quad-core system with three time partition

4.2.1.4 Implementation on big.LITTLE

The idea behind big.LITTLE is to provide a possibility to migrate running tasks and even the operating system, between the more efficient Cortex-A7 cluster to the high performance Cortex-A15 cluster depending on current demands. In this context, the cluster that is going to be set into standby is referred to as the "outbound cluster". The cluster that is going to be set into a state of operation is referred to as the "inbound cluster". There are also possibilities to run both clusters simultaneously, and thus, increasing the number of cores in a system to eight.

In this section we describe applications of the presented algorithm on the ARM big.LITTLE architecture. In general, it is a system integrator’s decision whether a particular VM/thread should be event or time triggered, as well as to define the tolerate-able delay for processing external events with real-time requirements. Thus, for the rest of this section we assume that system integrator has classified all VMs/threads in the system.

During normal operation the proposed algorithm works on big.LITTLE architecture as on any other SMP system. The interesting cases arise due to the switching between clusters as well as powering up and down a cluster. In the following we discuss these three cases and how they affect timing of the scheduling algorithm.

4.2.1.4.1 Cluster migration

Cluster migration introduces a new delay into the system. Let us call this delay *t_switch*. Since the amount of data which is needed for cluster migration is always the same (the content of CPU registers) the delay is deterministic. The decision to make is *when* to trigger this switch to minimise/mitigate negative effect on the system behaviour.
There are two possibilities:

1. Switch can happen at any time
2. Switch happens at predefined points in time, e.g. during time partition selected by a system integrator

In the first case the reaction time for incoming external events will be increased, in the worst case, by the time needed for a cluster switch. From the application point of view, running in a current time partition, the corresponding time slot will be “shortened” by the time needed for a cluster switch.

There is one corner case at the time partition boarder: when time partition is almost used up, and its rest time (let us call it *t_rest*) is less than the switch time, then the current partition will be extended by *t_switch – t_rest.* Thus, the following time partition will experience a jitter with the maximum bounded *t_switch*. Thus, allowing the switch at any point in time will introduce an additional jitter into the system behaviour.

In the second case, system integrator defines which time partition is used for the switch, e.g. system integrator integrates an application, which is responsible for cluster switch, into a specific time partition. By a proper selection of the time partition duration and point of switch, the system will not experience any non-deterministic effects.

4.2.1.4.2 Cluster power-up/down

In both power-up and power-down scenarios the system integrator has to define the schedule for the maximum number of CPUs available in the system. Since the base of the scheduler is a time-triggered scheduler, system integrator has to define a scheduling schemata (or major time frame), for every combination of active CPUs he plans to employ in the system. In the case of big.LITTLE there should be three major time frames. Note that PikeOS supports switching of major time frames during runtime.

The power-up of a cluster introduces an initialization delay of the CPUs (let us call it *t_up*) and a delay for switching scheduling schema (*t_sch*). Thus, the overall delay will be *t_up*+ *t_sch.*

The power-down of a cluster introduces several delays:

- Switching scheduling schema (*t_sch*).
- Migrating threads from CPUs to be turned off (*t_down*)
  - saving CPU and thread contexts
  - handling sleeping threads
  - forwarding interrupts to online CPUs

Similar to the switch of clusters, it is better to make a switch in fixed time-partition to mitigate arbitrary system level jitters. Since, delays for power up/down a cluster are longer than for the switch, the effect on the system behaviour, especially on time critical tasks, will be severe. Thus, only non critical tasks should be placed/moved on the dynamic cluster.

4.2.2 Approach 2: Extending the CFS scheduler for heterogeneous platforms

One of the purposes of the KVM hypervisor, is to take advantage of the well tried and tested infrastructure of the Linux kernel (e.g. memory management, scheduling, etc.), instead of re-implementing them. Thus, one could say that KVM turns the whole Linux kernel into a full blown hypervisor.

In the KVM/QEMU paradigm, each VM is essentially a POSIX process along with a number of threads (QEMU book keeping, VCPUs, etc.). For this reason there are no scheduling algorithms implemented inside KVM, mainly because the spawned VM is a regular application process that will be scheduled transparently by the kernel.

The main problem arises when heterogeneous systems enter the picture. The Linux kernel doesn’t have a mechanism to differentiate between different CPU types, and its scheduling capabilities are optimized mainly for symmetric multicore systems. With the introduction of big.LITTLE a great effort was created to efficiently support such heterogeneous architectures, which focuses on extending the Completely Fair Scheduler (CFS).

4.2.2.1 CFS scheduler limitations

The limitation of the CFS scheduler on heterogeneous platforms comes from the fact that its scheduling algorithm doesn’t take into account the CPU time that is consumed by each task. Basically load-balancing is expressed by the internal CPU load measurement as follows:

$$cpu_{load} = cpu_{power} \sum_{task} priority_{task}$$

Figure 7: CPU load calculation on CFS scheduler

While CFS can handle the distribution of tasks fairly evenly (assuming basic kernel support for big.LITTLE), this is not desirable if power efficiency is the target. A Cortex-A15 core, while being approximately twice as fast as a Cortex-A7 core (clock for clock basis), it is also 3 times worse in power consumption. For this reason it is appropriate that the scheduling algorithm in the kernel must take into account the differences of the CPUs included in the system.

4.2.2.2 big.LITTLE kernel scheduling

Due to power efficient constraints, the most important aspect of scheduling in a big.LITTLE system is to take advantage of the low power consumption of LITTLE cores. With this in mind we can enforce a set of strict policies to achieve maximal results as follows.

- All tasks should be bound on LITTLE cores unless:
    - The overall LITTLE core load is above a certain threshold, and
    - Task priority is set to high by the user or system (nice value).

The goal of this policy is to utilize the big cores as less as possible to reduce power consumption, but not restrict or affect the maximum performance of the system:

- Low intensity tasks that are frequently used (daemons, background processes, etc.) will not suffer by being bound on a LITTLE core.
- Low priority tasks with high intensity are not needed to be in big cores as they are not critical to finish early.
- Migration of tasks can be dynamically adapted by current constrains and requirements.

In order for these rules to be met, the scheduler must know of the asymmetry of the platform. For this reason a scheduling domain is created for each type of CPU in the system, two for the case of big.LITTLE, but theoretically more can be added. Tasks are placed in their respective cluster every time the load history is changed, according to the load thresholds being set.

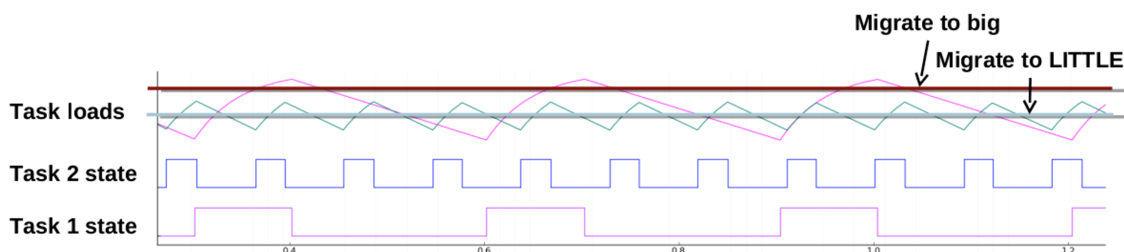

Figure 8: Migration of tasks depending on load

4.2.2.3 Power efficient scheduling and monitoring metrics

For maximizing power efficiency, there are of course other features in the Linux kernel, beyond strict scheduling. As most modern architectures, big.LITTLE (Cortex-A15 & Cortex-A7) also has a range of operating frequencies and voltage levels that can be used to further enhance performance, depending on the power constraints and the user requirements.

Currently the Linux kernel has the following mechanisms:

- cpufreq
- cpuidle

The cpufreq interface is essentially a list of operating frequencies and voltages coupled with a policy called governor that handles the way that frequency and voltage scaling will occur on the system. The kernel has a range of governors that can be used in any specific time, based on the user needs (performance, powersave, userspace, ondemand and conservative). With a proper cpuidle driver we can achieve tremendous power reductions when the system is idling, by selecting the proper “C state” sleep, and avoid wasting cycles on aggressive state toggling. Combining these features with the extensions of the CFS scheduler we can achieve a level of power granularity that is not possible with traditional SMP setups.

In order to capture and see the effects of scheduling and power management features, we have implemented an application for the purpose of monitoring various system metrics in real time. Measurements such as frequency, voltage, current, CPU load, etc. are needed to better assess changes in scheduling. The application relies on a lightweight daemon on the target, which samples all the needed metrics and then exports samples in order to be plotted and seen by the user. Captured data can then be compared and used to fine-tune the behaviour of the system.

5 Conclusion

We presented two approaches for scheduling on heterogeneous platforms.

The first approach focuses on mixing time- and event-triggered tasks. On SMP system it can be achieved more efficiently by assigning event- and time- triggered tasks to different CPUs. Despite this fact the system integrator is still have to decide *when* to allow usage of big.LITTLE features. Purely dynamical usage (e.g. triggered arbitrarily by non-critical graphics applications) can violate timing requirements for both time and event triggered tasks. Thus, it could be possible to achieve main qualification requirements (determinism, responsiveness). The main functional requirements (events prioritisation and affinity) overlaps with SMP design of PikeOS. In case of PikeOS, where all programs are sorted out to partitions (also know as virtual machines), the threads shall not cross partitions boundaries. Otherwise, it may interfere with mixed-criticality design of the overall system. Therefore, dynamic migrations and mutual exclusion should be implemented in user-level scheduler. PikeOS provides necessary support (e.g. manipulation of threads data structures). To satisfy “common cache control” requirement, the hardware should support it. In case of vIrtical platform it holds automatically. The non-functional requirements “reallocation of unused time” can be easily achieved with background partition. The “audit” requirement is already implemented in PikeOS health monitor and is not considered in this document. Thus the presented algorithm suits well for mixing event and time-triggered tasks on the vIrtical architecture. This judgement, however, holds only with respect to the determinism and the responsiveness of a single homogeneous cluster.

The second approach is based on existing KVM/Linux infrastructure and leverages power efficiency with the help of ARM big.LITTLE. The focus is on generic applications with event triggered semantics which can tolerate timing fluctuations in execution. Non deterministic requirements can be met (auditing, affinity, reallocation of unused time) due to the variety of available tools and maturity of the Linux kernel. Furthermore, power efficiency and dynamic migration is achieved, by extending the kernel’s scheduling infrastructure. We think that for generic applications and with suitable runtime monitoring techniques (e.g. see Deliverable D3.4) it is possible to introduce power-aware events into systems based on the big.LITTLE architecture.

Outcast: Despite the fact that the embedded systems are the first one to profit from such architectures, additional case-studies on security aspects (e.g. using it in smartphones with trusted and non-trusted applications, mixed-criticality applications) has to be carried out.