Higher Utilization of Multi-Core Processors in Dynamic Real-Time Software Systems

Thomas Hanti, Michael Ernst, and Andreas Frey
Technische Hochschule Ingolstadt, Esplanade 10, 85049 Ingolstadt
Email: {Thomas.Hanti, Michael.Ernst, Andreas.Frey}@thi.de

***Abstract*—The number of functions and complexity in real-time Electric/Electronic systems is constantly increasing. With the ongoing electrification of vehicles an increasing number of software functions is expected to be integrated in the Electric/Electronic systems. In order to provide the necessary calculating power, more and more multi-core processors will be used in Embedded Electronic Control Units. With the rising number of functions on multi-core processors dynamic software systems can help to achieve a more efficient utilization than currently used static system configurations. Therefore the step from static to dynamic system configuration will be the key. Our paper will present the design of a new scheduling approach, the Hierarchical Asynchronous Multi-Core Scheduler (HAMS), for real-time Electronic Control Units. Special strategies for dynamic system design and dynamic software system description will be presented as well as a first evaluation of our design.**

***Index Terms*—real-time scheduling, scheduler design, dynamic scheduling, hierarchical scheduling, asynchronous scheduling, multi-core scheduling, embedded scheduling**

I. Introduction

A. Issues with Multi-Core Systems

The increasing demand of calculation power in the desktop and server environment has urged the processor manufacturers to either increase the clock speed or to add more cores to a processor. As the maximum clock speed of processors is running up against its physical limits, multi-core processors have gained significant importance. Nowadays multi-core processors have become the standard architecture for applications where a lot of calculation power is needed, even in the embedded domain.

As multi-core processors are widely used in non real-time applications in the consumer electronics market, their applications are expanded to safety critical real-time application in cars or airplanes. New automotive technologies like car to car communication, highly automated driving, camera based driver assistance and infotainment are current topics that push high performance embedded devices in the car. In the context of avionics the operation of unmanned aerial vehicles with autonomous situation interpretation using multiple sensors and cameras and autonomous decision making in complex scenarios also need high performance embedded devices. This motivates the use of multi-core processors which are able to compute safety relevant real-time tasks.

However, even if there are numerous multi-core processors for control units in cars and airplanes available, an overall concept for an efficient utilization is not applicable. Today's multi-core platforms are used with a static task assignment. This means that each real-time task is assigned to one core of the multi-core system in an apriori engineering process that requires a detailed analysis, e.g. the exact Liu-Layland test to check if the real-time task set will meet all its deadlines. Still the static assignment of tasks does not exploit the full potential. The industry demands solutions to use the potential even further.

Our analysis has encountered two main limitations with this approach:

- Static assignment of tasks in multi-core systems generates inflexibility in system definition and usage. When a multi-core control unit is partitioned each task is statically assigned to one core reserving a part of the calculation power and memory. For example car features like cruise control, rain sensor or lane departure warning can be selected by the costumer at the time of order. But the corresponding hardware and software system in the actually built car is statically defined and tested. Each alternative system configuration requires extensive verification and testing. As a consequence of cost evaluation the system is statically defined taking inefficient usage of the processor into account, leaving a high potential for further cost and efficiency optimization.
- Real-time embedded systems are designed according to maximum execution times. In order to keep the carefully balanced timing behavior untouched the processor resource is statically allocated. The calculation time of tasks, especially in driver assistance systems and systems for highly automated flying, is increasingly dependent on the situation. Along with the rising percentage of those functions in future cars or unmanned aerial vehicles this leads to a non predictable timing behavior in statically assigned systems. As it is impossible to foresee and test all possible situations an onboard balancing can improve the usage of the processor resource.

Manuscript received June 28, 2013; revised September 2, 2013.

doi: 10.12720/ijoee.1.4.249-255

In order to further optimize the usage of the embedded real-time multi-core systems we propose a new approach in multi-core real-time scheduling. Using a dynamic scheduling environment which will allow us to utilize the processor more efficiently than static scheduling reducing costs and improve efficiency.

B. Related Work

Single-core real-time scheduling is by far the most precisely examined area of scheduling strategies due to its relevance for safety. Important real-time scheduling algorithms are Rate Monotonic Scheduling (RMS) for static scheduling, Earliest Deadline First (EDF) and Maximum Urgency First (MUF) [1] and [2] for dynamic scheduling.

Liu and Layland et al. have proven in their static RMS research that the maximum utilization of a processor is 69% (ln2) for W_∞ tasks where all tasks can meet their deadlines [3]. Other research has demonstrated that the maximum utilization for W_∞ tasks can be much better by using other schedulability tests like Hyperbolic Test [4], Time Demand Analysis or Pillai-Shin Test [5], but the calculation time, task requirements, effort and calculation power for such tests is much higher than the classic Liu and Layland analysis.

A dynamic scheduling algorithm is the MUF algorithm which improves Liu and Layland's EDF algorithm. This dynamic scheduling algorithm can have up to 100% processor utilization on single-core systems.

Subsequently RMS and EDF algorithms have been adapted to multi-core systems. These adaptations led to different runqueue approaches as seen in Fig. 1.

The partitioned approach uses a separate runqueue for each core. The runqueues are filled with a set of tasks and then managed by each core according to the underlying scheduling algorithm on its own. In the partitioned real-time approach the tasks do not migrate from one core to another instead they are statically distributed beforehand. The partitioned approach equals the algorithms used for single-core processors and brings along the same disadvantages as address in section 1A.

Another approach is the global single "global" runqueue for the whole system. Whenever a core has finished task calculation, it is going to pick the next one out of the global runqueue [6] and [7]. The tasks will therefore be moved freely from one core to another. Particularly in EDF this approach is common because load balancing among cores is improved. But it shows significantly less processor utilization than the partitioned approach [8]. Fig. 1 lists common scheduling algorithms and their utilizations in comparison to our HAMS scheduler.

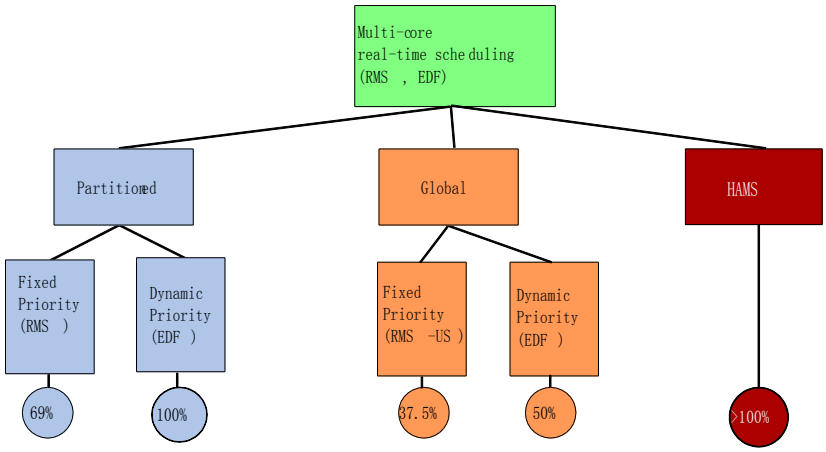

Figure 1. Design space of Multi-Processor Real-Time Scheduling [9] extended with the Hierarchical Asynchronous Multi-Core Scheduler (HAMS)

C. Motivation and Structure

The research in real-time scheduling and multi-core real-time scheduling is very intense. But none of the so far applied scheduling techniques, like RMS, EDF, global or partitioned runqueues are able to provide the required performance enhancements in the automotive and avionic industries. All common solutions are designed for static real-time systems, leaving further optimizations regarding efficiency of the processor usage.

In this paper we introduce a dynamic asynchronous multi-core real-time scheduling design (HAMS), a new approach in multi-core scheduling; which:

- allows an optimal usage of the processor in all situations
- is able to load balance the workload in a real-time environment
- introduces new power saving features like dynamic voltage frequency scaling(DVFS) into real-time task planning
- can be used with asymmetrical multi-core processors

Thus improvements in cost and efficiency of processor usage will be the result!

This paper is structured as follows: Section II will introduce the basic design and the knowledgebase for our Hierarchical Asynchronous Multi-Core Scheduler (HAMS) consisting of a task-, logical linkage- and system model. In Section III *A* we will introduce the different parts of our HAMS scheduler in more details and compare it with today's statically scheduling III *B*.

In Section IV we will outline our expectations on the hierarchical asynchronous scheduling concept and the

issues that come along with hierarchical asynchronous scheduling rounded up with an outlook in Section V.

II. Design of the HAMS Scheduler

A. Basic Design of the Hierarchical Asynchronous Multi-Core Scheduler (HAMS)

To overcome the limitations that come along with static multi-core configuration a more general approach is needed that allows the use of all cores with an overall strategy. As described in section 1A different task types, consisting of hard, soft or variable deadlines, can cooperate on one system. To balance and treat these different types correctly it is very important that our scheduler is hierarchically structured. With a Second Layer Scheduler (SLS) controlling the task distribution and a separate asynchronous First Layer Scheduler (FLS) managing the local runqueue for each core (see Fig. 2).

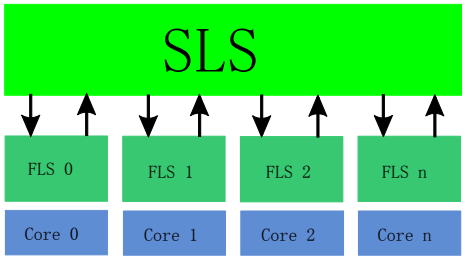

Figure 2. Basic HAMS layout

The HAMS SLS scheduler is the highest entity of the HAMS scheduler which controls the correct and efficient distribution of tasks among the cores taking the current system state and task timing information into account and communicates with the lower level.

The HAMS SLS scheduler incorporates a predefined global knowledgebase consisting of all tasks running on the system, the task-dependencies and a description of the system itself, i.e. a task-, logical linkage- and system model. With this knowledge the HAMS SLS scheduler is able to control the tasks in the system, i.e. to dynamically suspend tasks, migrate tasks among cores and delay or shift kernel administration tasks to increase calculation power by simultaneously decreasing power consumption and still comply with all deadlines, i.e. utilizing the system more efficiently. The multi-core task model for the system is the basis for the hierarchical design and the knowledgebase.

The lowest entity of the HAMS scheduler, the HAMS FLS, is able to flexibly integrate tasks that are assigned from the HAMS SLS scheduler to its local runqueue. All FLS schedulers are running asynchronously in order to fully exploit the multi-core potential. The local workload calculation of the HAMS FLS scheduler is the basis for the SLS task distribution decision. The HAMS FLS scheduler utilizes already existing scheduling classes like RMS or MUF. These classes will be the same as used in today's schedulers. In respect to the scheduling class the lowest entity observes the task deadlines and hence task calculation sequence as any normal real-time scheduler.

B. A Multi-Core Task Model

A common model to describe real time tasks is the standard task model for periodic real-time tasks $\tau_i = \{C_i, T_i, D_i\}$ where the parameters of a task τ_i are represented by its worst case execution time C_i, its period T_i and its deadline D_i. A task set τ is expressed with $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ [9]. For periodic real-time tasks with matching period and deadline, this representation reduces to a 2-tuple $\tau_i = \{C_i, T_i\}$ [10]. Sporadic real-time tasks are characterized by the 3-tuple $\tau_i = \{C_i, A_i, D_i\}$ where A_i is the minimum separation between two calculation sequences of the same task, where $A_i > 0$. In an aperiodic task the same 3-tuple as described for sporadic tasks can be used but here A_i can be zero [11].

For our HAMS SLS scheduler this task description is not detailed enough. First of all multi-core processors can successfully vary their clock frequencies stepwise to reduce power consumption, e.g. by dynamic voltage frequency scaling [12]. And by reducing the clock frequency the worst case execution time will become longer because a task will need more time to calculate its results. So the worst case execution time for each possible processor speed ($C_{i\,max}, \ldots, C_{i\,min}$) has to be added to the task[13].

As the HAMS SLS scheduler will make use of different scheduling classes, like RMS and MUF, the attribute criticality level CR_i of a task has to be added by the user. With the CR_i value the HAMS scheduler will make sure that no critical tasks will fail during failure situations whatever scheduling class is used on the core. But when different scheduling classes are involved, each task has to be assigned to one specific scheduling class SC_i. In our HAMS scheduler it is possible to use almost every already existing scheduling classes, like MUF, RMS, single shot and even fair priority scheduling.

Most publications for real-time multi-core scheduling assume they are in a homogeneous, symmetric environment. For example in the P4080 processor not every core in a real-time embedded control unit has access to the full set of peripherals [14]. Therefore a value to which cores the task is bound has to be added called: CB_i. A task can be bound to one core, two or more cores as well as being free from any core bound. Further on, a task needs to have an attribute of its required peripherals NP_i. For example this may be an Ethernet, Serial Interface or a Floating-Point Unit.

The last, but most important attribute of a real-time task in the HAMS scheduler, is the task calculation state TC_i. With the help of this attribute a task can prolong or shorten its parameters (C_i, T_i, D_i) when there is a reason to do so and signal it to the HAMS scheduler. Additionally the attribute TC_i of one task can be linked with another task, called a logically linked task set (explained later in section 2C). Particularly this attribute is very important because it helps the scheduler to increase the dynamic behavior, introduce planning essentials for DVFS and increase efficiency of the overall system by simultaneously ensuring schedulability. By the help of (1) the multi-core task model of a periodic real-time task for RMS and MUF scheduling classes can be summarized.

$$\tau_i = \{CR_i\ ;\ SC_i;\ NP_i\ ;\ CB_i\ ;\ \{TC_m\}_i\ \}$$
$$TC_m = \{TC_1, \dots, TC_n\}$$
$$TC_n = \{D_n;\ T_n\ ;\ C_{n\,min}, \dots, C_{n\,max}\} \quad (1)$$

Due to the fact that the tasks are located in real-time embedded system the first four parameters of (1) should be static and thus kept unchanged during runtime. But depending of the TC_i the period time T_i , the deadline D_i and the worst case execution time C_i can vary. Equation (2) adapts the TC_i value for sporadic or aperiodic tasks in RMS and MUF scheduling classes.

$$TC_n = \{A_n;\ T_n\ ;\ C_{n\,min}, \dots, C_{n\,max}\} \quad (2)$$

The knowledgebase is the backbone of our system. The design uses the multi-core task model which is summarized in (1). It is important that, additionally to the already described parameters, every task is assigned with a global identification so that the underlying operating system is able to recognize the task in a running environment.

C. A Task Dependancy Model

In an asynchronous dynamic system architecture where tasks can migrate between different cores the task dependency model is a key issue for efficient assignment. Task dependencies express temporal dependencies between tasks, for example: one task needs to run before another task (consecutively), they need to run at the same time (concurrently) or a task can only run when another task is also running.

One common technique in real-time scheduling is to assign a higher priority to the task which needs to run before another one. This technique will always work, on a single-core processor with a preemptive RMS scheduling algorithm.

On a multi-core-system it may happen that one task is assigned to one core and the other task is assigned to another core so that both tasks are not running on the same core with inexplicit consecutive order. While both tasks have the highest priority in each set of tasks on the particular core, it occurs that both tasks run concurrently even if they are supposed to run consecutively. A possible solution is to statically assign these two tasks to the same core of a multi-core system or to introduce special multi-core inter-task communication techniques. But this will lead to negative effects in load balancing and thus to increased power consumption [15]. In the knowledgebase of the HAMS SLS scheduler such tasks are grouped together in logically linked task sets. An example of the automotive industry clarifies the usage of the task calculation state TC_i .

In a modern car, two driver assistant systems, the cruise control and the park distance control feature can be implemented. The cruise control is realized with one task, the "cruise_main (cm)" task, in charge for controlling the vehicles speed when activated. The park distance control consists of two implemented task, the "park_main (pm)" task in charge of acoustically notifying the driver and the "park_edge_detection (ped)" task which calculates the distance to the next obstacle. In Table I the TCs and the corresponding execution times and period are listed for periodic every task, where period time equals execution time. With the requirement that the cruise control can only be active when the speed is above 30km/h the three tasks can be logically linked (Table II). For example when the cruise_main task is in state TC_{cm1} then and only then the park_main can be in TC_{pm2} or park_main and park_edge_detection are in the active state (TC_{pm3}; TC_{ped3}). With this help load balancing and therefore the power consumption becomes more efficient.

TABLE I. TASK CALCULATION STATE EXAMPLE

Task name	TC for tasks in different situations		
Event→	speed<30km/h	speed>30km/h	Activated
cruise_main	TC_{cm1} =(1;5)	TC_{cm2} =(2;4)	TC_{cm3} =(3;5)
park_main	TC_{pm1} =(1;5)	TC_{pm2} =(1;5)	TC_{pm3} =(1;3)
park_edge_de	TC_{ped1} =(0;3)	TC_{ped1} =(0;3)	TC_{ped3} =(1;5)

TABLE II. LOGICAL LINKAGE AMONG TASKS

Task name	Logical linkage among tasks		
	speed<30km/h	speed>30km/h	Activated
cruise_main	$TC_{cm1} \leftrightarrow$ ($TC_{pm2} \wedge$ ($TC_{pm3} \vee TC_{ped3}$))	$TC_{cm2} \leftrightarrow$ ($TC_{pm1} \vee TC_{ped1}$)	$TC_{cm3} \leftrightarrow$ ($TC_{pm1} \vee TC_{ped1}$)
park_main	$TC_{pm1} \leftrightarrow$ $TC_{cm1} \vee TC_{ped1}$)	$TC_{pm2} \leftrightarrow$ ($TC_{cm2} \wedge TC_{cm3}$)	$TC_{pm3} \leftrightarrow$ ($TC_{cm1} \vee TC_{ped3}$)
park_edge_de			$TC_{ped3} \leftrightarrow$ ($TC_{pm2} \vee TC_{cm1}$)

D. A Multi-Core System Model

When all parameters and linkages of the real-time tasks have been identified the knowledgebase for the HAMS SLS scheduler needs to be completed with a system model. As tasks can be dependent on peripherals the HAMS scheduler needs to know which core has access to which peripheral device. In this system model each core σ_i must be described with the parameters: The steps from minimum to maximum clock frequency CF_i , minimum and maximum memory address MA_i and available peripherals AP_i as in (3). With these parameters the HAMS SLS scheduler can load balance tasks correctly, i.e. which core has access to which peripheral or memory address and react appropriately to failures like loss of peripherals.

$$CF_i = \begin{Bmatrix} CF_{i\,min}, \dots, CF_{i\,max}\ ;\ MA_{i\,min}, \dots\ MA_{i\,max}; \\ AP_{i\,min}, \dots\ AP_{i\,max} \end{Bmatrix} \quad (3)$$

E. Knowledgebase Creation

When all parameters and linkages of the real-time tasks have been identified the knowledgebase for the HAMS scheduler can be designed in an offline tool. In this tool the system model has been entered into the knowledgebase, all real-time tasks that need to run on the system must be entered as provided in (1) and (2) as well as their logical linkages. Therefore all parameters (C_i ; T_i ; D_i) and linkages have to be defined. Especially in the HAMS scheduler the attribute TC_i of every task makes this very difficult. Here the values C_i , T_i , D_i for every TC_i have to be detected. An offline tool is used to check the correctness, i.e. feasibility and schedulability, and the optimal load balancing of the system upon the

entered parameters. The output is stored in a file which will be integrated in the online scheduler.

III. Hierarchical Asynchronous Multi-Core Scheduling

A. Initializing the Layered Structure in the HAMS Scheduler

The HAMS scheduler startup sequence is defined by the initial startup sequence of the operating system (OS) as the scheduler is a task that is initialized as part of the OS. In the description we assume a LINUX kernel as operating system.

- *FLS with normal operating sequence:*

In the first seconds of the system booting procedure the FLS initializes its parameters and the communication interface to the SLS. When this is done the kernel initializes system internal tasks and the SLS. While the system boots up, the FLS will run the default scheduling class of the OS.

Once the OS has finished its initialization, first the SLS and then all the real-time tasks (representing the features of the embedded system) are spawned. Right after completion of the spawning for all real-time tasks, the initialization and boot up sequence is done.

Afterwards each core transmits information about which tasks are currently running on it and the task-structure for each task to the SLS. Additionally the FLS will get a list of tasks and the corresponding scheduling class the core has to schedule. It picks the next task to run, manages the task deadline miss handling, the OS internal periodical system timer, the frequency of its core, keeps alive the communication with the SLS and obeys to migration calls from the SLS (see Fig. 2).

- *SLS normal operating sequence:*

After the SLS has been spawned by the FLS it will read out the knowledgebase which is stored inside the kernel since the compile time. By doing this the SLS will know everything about the task dependencies, the system model and the tasks that should run on the system.

After this, the SLS will establish the communication with the FLS and receive a list of tasks that are running on the system and stores them in a global runqueue. The SLS will start to calculate the correct distribution of the real-time tasks among the cores and send the results to the FLS. The SLS now keeps the communication with the FLS alive, surveils their behavior, reacts to threads like unschedulable task sets, communication loss to one FLS and system failures e.g. loss of peripherals. The responses of the SLS to failures are to migrate the task onto another core of the system, if possible, or to command the whole system into a failure state where only the tasks with the highest criticality level are scheduled.

B. Comparison of HAMS Scheduler Design

To validate the design of the HAMS scheduler against the existing static scheduler design an example illustrates the HAMS advantages. We assume that the tasks introduced in the Tables I and II of section 2C are statically scheduled on one core. We must assume that all three tasks can be active together, because static scheduling does not distinguish between any logical linkages. Thus the maximum allowed utilization in a single core system is exceeded by the static scheduler using for calculation. Hence in a statically scheduled system we would need a dual core system. In HAMS scheduling it is possible to schedule all three tasks on one core, because the HAMS scheduler can assure that either the speed control is active or the park distance control is active. This feature of the HAMS scheduler allows us to utilize the processor far more efficiently as in comparison to static scheduling. Equivalent to static scheduling we can reach more than 100% processor usage as described in Table III.

TABLE III. Static Vs. Hams Scheduling on A Single Core

speed v in km/h	Comparison					
	static scheduling			HAMS		
	Tasks	W_{max}	W_{is}	Tasks	W_{max}	W_{is}
v>30	3	73%	113%	2	88%	80%
v<30	3	73%	113%	3	73%	73%

When using a multi-core system, e.g. a dual core, a statically scheduled system is forced by default to split the tasks cruise control and park distance control on two separate cores. Thus the schedulability checks will work. As a result the task distribution will be affected, i.e the system is not allowed to migrate the cruise control and park distance control any more. Hence an increase in power consumption will be the result as illustrated in table IV, based on the formula $g(S) = S^2$ from [16]. The knowledgebase enables the HAMS scheduler to actively load balance the workload or even shut down a core depending on the situation.

TABLE IV. Static Vs. Hams Scheduling Power Consumption

speed v in km/h	Comparison					
	static scheduling			HAMS		
	Core	W_{is}	P	core	W_{is}	P
v<30	1	20%	0.04	1	40%	0.16
	2	53%	0.28	2	33%	0.11
result		∑	0.32		∑	0.27

A further advantage of our HAMS scheduler is the usage of processor power saving features, e.g. dynamic voltage frequency scaling (DVFS). In the HAMS scheduler the runtimes for a real-time task according to all possible CPU frequencies is listed in the knowledgebase. On this basis the HAMS scheduler can decide to throttle or speed up a single core without intense pre calculations like in today's static scheduling. By looking at table IV the HAMS scheduler can decide to lower the frequency of core 1 by 50%. The result will be ca. a 50% higher utilization on core 1. Taking a linear rise and fall of the power consumption according to the workload the HAMS advantages can be illustrated. Fig. 3 and Fig. 4 show that the overall power consumption with the HAMS scheduler (12,6W in Fig. 4) is lower than static scheduling (14,6W in Fig. 3) by using DVFS. Thus the HAMS scheduler cannot only make use of DVFS to

improve the system efficiency, it can do it in a real-time system environment without any violation of the deadlines.

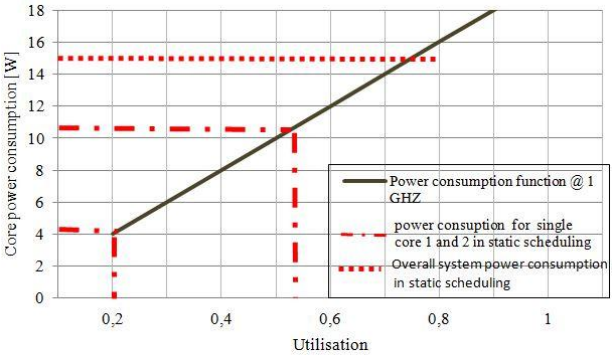

Figure 3. Static scheduling without DVFS power consumption =14,6W

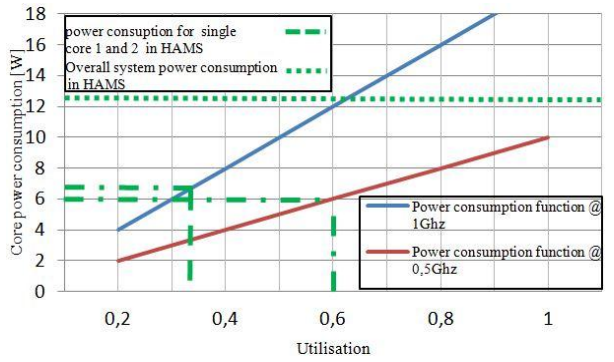

Figure 4. HAMS scheduling with DVFS power consumption =12,6W

Another advantage of HAMS scheduling is the failure mode. When failures in the system occur the HAMS scheduler can assure that critical real-time tasks, marked with the highest CR_i, will not fail. Or tasks that are also impacted by the failure will not be scheduled. In a static system the system response to failure is limited.

IV. Reasearch Topics in Hams Scheduling

By the help of HAMS scheduling the step from statically to dynamic systems can be made. But this advantage comes along with three known issues that have to be addressed. The first issue is to determine the precise values of the multi-core task model for each task. Exact values can only be measured with extensive testing. Thankfully various tools for testing a real-time task runtime under multiple circumstances exist.

The second issue is based in the real-time task distribution process of the SLS. Distributing tasks among cores with as many parameters as mentioned above in the multi-core task model is a very time consuming and a NP-Hard process. Especially when the schedulability of all real-time task sets for every possible configuration has to be checked during operation (online). In a real-time system such proceedings have to be as fast as possible, because the tasks shall not miss their deadlines. Hence an online approach like this will cause timing issues. To speed up the process of finding a correct task set, the knowledgebase has to be extended with a distribution model.

The last issue in HAMS scheduling is related to the communication process. When the SLS communicates with the FLS it has to be secured that no other program can disturb, block or use the connection for its own purpose. If the connection can be modified by other programs the correct functionality of the SLS cannot be guaranteed leading to an unstable system with unknown results.

V. Conclusion and Further Reasearch

The HAMS scheduler shows advantages in comparison to normal static scheduling. The next logical step is to test this design in practice. We will use a multi-core ARM Cortex™ A9 platform which can run a basic Linux based operating system. In the current Linux kernel we will replace the completely fair scheduler with our HAMS scheduler. By doing this we will modify the scheduling classes already existing in the Linux kernel and insert new classes for RMS, MUF and single shot scheduling to complete the FLS. In addition the SLS has to be built with a special focus on algorithms for fast real-time task distribution and secure inter-scheduler communication.

When the system is completed, we will validate the systems behavior. This will let us draw conclusions about the promised efficiency increase. For validation proposes we will use different tasks that will simulate specific behaviors that run in an automotive embedded control unit.

The result is an overall validation of the advantages and disadvantages of a HAMS scheduler design and the underlying algorithms.

References

[1] D. B. Stewart and P. K. Khosla, "Real-time scheduling of dynamically reconfigurable systems," in *Proc. IEEE International Conference on Systems Engineering*, Dayton, OH, 1991, pp. 139 - 142.

[2] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. the IEEE Real Time Systems Symposium*, Santa Monica, CA, 1989, pp. 166 - 171.

[3] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[4] E. Bini, G. Buttazzo, and G. Buttazzo, "A hyperbolic bound for the rate monotonic algorithm," in *Proc. 13th Euromicro Conference on Real-Time Systems*, Delft, 2001, pp. 59 - 66.

[5] P. Pillay and K. Shin, "Real-time dynamic voltage scaling for low power embedded operating systems," in *SOSP 01: Proc. the Eighteenth ACM Symposium on Symposium on Operating Systems Principles*, vol. 1, Oct. 21-24, 2001, pp. 89-102.

[6] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," presented at International Parallel and Distributed Processing Symposium, Phoenix, AZ, April 22-26, 2003.

[7] L. Lundberg, "Analyzing fixed-priority global multiprocessor scheduling," in *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, 2002, pp. 145-153.

[8] T. Baker, "A comparison of global and partitioned EDF schedulability tests for multiprocessors," presented at International Conf. on Real-Time and Network Systems, Paris, Nov. 8-9, 2005.

[9] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Proc. 21st Euromicro Conference on Real-Time Systems*, Dublin, 2002, pp. 239 - 248.
[10] N. Guanyz and M. Stiggey, "Fixed-priority multiprocessor scheduling with liu \ layland's utilization bound," in *Proc. Real-Time and Embedded Technology and Applications Symposium*, Stockholm, 2010, pp. 165 - 174.
[11] R. Mallreceived, *Real-Time Systems: Theory and Practice*, 1st ed. India, Addison Wesley, 2007, ch. 3.
[12] Texas Instruments Inc., *OMAP4430 Multimedia Device*, 2nd rev. Texas Instruments, Dallas, TX, 2013, ch 3.
[13] D. Zöbel, *Echtzeitsysteme : Grundlagen der Planung*, 1st ed. Berlin, Springer, 2008, ch. 4.
[14] Freescale Semiconductor Inc., *P4080: QorIQ P4080 Eight-Core Communications Processors with Data Path*, 1st rev. Freescale Semiconductor Inc., Tokyo, 2013, ch. 6.
[15] T. AlEnawy and H. Aladin, "Energy-aware task allocation for rate monotonic scheduling," in *Proc. the 11th IEEE Real Time and Embedded Technology and Applications Symposium*, San Francisco, CA, 2005, pp. 213 - 223.
[16] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," presented at Parallel and Distributed Processing Symposium, Nice, April 22-26, 1003.

Thomas Hanti, 1987, born in Ingolstadt and studied International automotive engineering (Master) at the university of applied sciences Ingolstadt from 2011 to 2012. Since 2012 he is Ph.D. student at the Technische Hochschule, University of applied sciences Ingolstadt in cooperation with the Technical University Chemnitz and CASSIDIAN in the research field of dynamic real-time embedded Systems and integrated modular avionics

Michael Ernst, 1987, born in Ingolstadt and studied Software Engineering for Embedded Systems (Master) at the university of applied sciences Ingolstadt from 2005 to 2011. Since 2012 he is Ph.D. student at the Technische Hochschule, University of applied sciences Ingolstadt in the field of dynamic real-time embedded Systems

Andreas Frey, 1969, born in Munich and has done his Ph.D. (Dr.-Ing.) from 1998 to 2003 at the University of the German armed forces, Neubiberg. In the years from 2003 until 2010 he worked in the department of Software-Engineering and Electric/Electronic Architecture for driving dynamics at the BMW AG. Since 2010 he is Professor for Aerospace Informatics and Avionics at the Technische Hochschule, University of applied sciences Ingolstadt

