Mapping a Multi-Rate Synchronous Language to a Many-Core Processor

Wolfgang Puffitsch, Eric Noulard, Claire Pagetti
ONERA - DTIM, Toulouse, France
{Wolfgang.Puffitsch,Eric.Noulard,Claire.Pagetti}@onera.fr

***Abstract*—This paper describes an end-to-end framework for the design and the implementation of real-time systems on a many-core architecture. The system, described in the multi-rate synchronous language PRELUDE, is translated into a set of communicating periodic tasks. We present a first heuristic to compute a partitioning of this task set that takes into account the specifics of the underlying platform, the Intel Single-chip Cloud Computer (SCC). In particular, the heuristic considers the communication between tasks. Furthermore, we provide a schedulability analysis that is used to validate the partitioning. We successfully apply the heuristic to several realistic use cases to evaluate the effectiveness of the proposed framework. For executing the task set, we have developed a run-time environment based on a bare-metal library that allows partitioned non-preemptive earliest deadline first (EDF) scheduling and manages the communication between tasks via the message passing mechanisms provided by the Intel SCC. The evaluation shows that the run-time overheads introduced by the framework are reasonably low.**

***Index Terms*—Real time systems, Multi-rate synchronous language, Many-core architecture**

I. INTRODUCTION

The ever growing performance demands of computer systems have driven the state of the art towards parallel computer architectures. The next generation of processors are many-core processors, which comprise numerous simple cores that are connected via an on-chip network. The shared memory model and cache coherence are replaced by explicit message passing.

For real-time systems, these many-core architectures provide new opportunities, but also introduce challenges [1], [2]. On the one hand, communication via an on-chip network is a potential source of unpredictability. On the other hand, the behavior of the relatively simple processor cores is easier to predict than of more complex multicore processors. Also, replacing the shared-memory paradigm with explicit communication avoids costly synchronization mechanisms. Therefore, we consider it worthwhile to explore the use of many-core processors for real-time systems.

A. Objective

This paper presents a framework that enables the execution of real-time systems described in a multi-rate synchronous language on a many-core platform. It translates the high-level specification and generates a mapping of real-time tasks to the underlying many-core platform. Furthermore, it provides a lightweight run-time environment for scheduling and execution of the resulting real-time system.

The framework supports task sets as generated by the multi-rate synchronous approach of PRELUDE [3]. The model comprises a set of periodic tasks that are connected by extended precedence constraints. These precedence constraints make it possible to express precedences between tasks at arbitrary rates. This makes the model expressive enough to cover a large class of real-time applications. Using a formal language for the description of dependent multi-rate task sets has also been advocated by Zeng and Di Natale [4] and Baruah [5]. The work presented in this paper does not necessarily require PRELUDE as input language and could be combined with other approaches that generate periodic task sets. However, the framework does not handle sporadic tasks.

The target platform for the presented framework is the Intel Single-chip Cloud Computer (SCC) [6], [7]. While some details of our framework are specific to that processor, we believe that the general principles described in this paper are also applicable to other many-core processors.

We argue that non-preemptive partitioned scheduling is an attractive solution. While we are aware that such an approach is not optimal with regard to schedulability, we believe that this is outweighed by the improved predictability, reduced overhead, and simpler implementation.

- *Partitioned scheduling.* On the one hand, the large number of cores on many-core processors reduce the importance of achieving high utilizations on individual cores. On the other hand, a case study by Brandenburg et al. [8] suggests that partitioned scheduling can schedule task sets with higher total utilizations than other policies when taking into account realistic overheads. While that case study evaluated a Sun Niagara platform, we expect the general trend will also hold on other many-core platforms.
- *Non-preemptive scheduling.* Non-preemptive scheduling simplifies worst-case execution time (WCET) analysis by eliminating the need for modeling preemption costs. While some of these costs are directly visible in the scheduler (e.g., the task switch time), the effects of preemptions on the hardware state and hence the WCET are more difficult to model [9]. Therefore, avoiding preemptions makes it easier to predict the system behavior. When considering the scheduler implementation, non-preemptive schedulers are simpler to implement than preemptive schedulers. Task switches are no more complex than function calls, and the use of asynchronous mechanisms such as interrupts can be avoided.

As scheduling policy we chose to implement earliest deadline first (EDF) scheduling. However, this could be easily replaced with fixed-priority or off-line scheduling, as long as the overall scheduling approach remains partitioned and non-preemptive.

B. Contributions

This paper describes the mapping of dependent real-time task sets to a many-core processor and a run-time environment that is suitable to execute such task sets.

1) Static Mapping: Given a periodic task set, the objective is to define a static mapping on the many-core platform. This consists in defining on which core each task executes and also mapping the communication buffers created by PRELUDE to the message passing infrastructure provided by the SCC. Due to the large number of tasks and cores, performing this mapping manually would be time-consuming and error-prone. Therefore, we automate the mapping process as far as possible.

We present a partitioning heuristic that takes into account the specifics of the real-time task set, the run-time system, and the underlying hardware. We detail a cost model for the SCC and describe a heuristic based on that cost model. The partitioning heuristic relies on the schedulability test for non-preemptive EDF scheduling of Fisher and Baruah [10]. While their schedulability test does not consider dependencies, we found that it approximates schedulability reasonably well when additionally bounding the load on each processor. To ensure the validity of the generated partitioning, the tool chain integrates a schedulability analysis for non-preemptive partitioned EDF scheduling.

2) Run-time Environment: The proposed framework provides a run-time environment that is based on a bare-metal framework and does not require an operating system. This avoids mechanisms such as interrupts and virtual memory, which are potential sources of unpredictability. Furthermore, the run-time system follows a "multiple instruction, multiple data" (MIMD) approach, where the created binaries are specific to particular cores. To support the communication between tasks, the run-time environment provides functions to access the message passing hardware mechanisms of the SCC.

We implemented a partitioned non-preemptive scheduler. The scheduler is tick-based, i.e., scheduling decisions are taken only at discrete instants of a chosen granularity, rather than at potentially every clock cycle. In order to cope with the imperfect synchronization of local clocks on the SCC and the non-negligible time for communication between cores, we propose to leave a gap between the end of a job's termination and the beginning of the next tick. The tick-based approach together with the gap enable the use of simple flags rather than semaphores for dependency handling.

On multiprocessors, non-preemptive scheduling of dependent task sets may be subject to *anomalies*, i.e., a feasible task set may become unschedulable if a task executes for less than its WCET. Our scheduler avoids these anomalies by enforcing that tasks do not terminate before their WCET.

C. Experimental Evaluation

As there is no sufficiently detailed timing model of the SCC, we conducted experiments to evaluate the overheads inferred

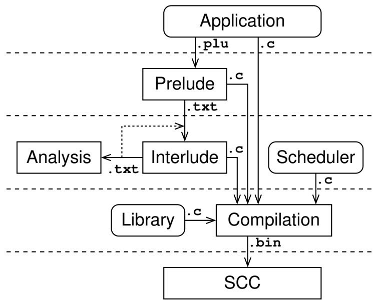

Figure 1. Tool flow

by our scheduling approach. This includes an evaluation of the tightness of clock synchronization as well as the time for communication over the on-chip network on the SCC. The results indicate that high contention can cause unfeasibly large communication delays and must be avoided.

Furthermore, we applied the mapping heuristic and the schedulability analysis to several examples. The examples are derived from industrial applications and comprise up to 375 tasks, with utilizations up to 10.34. The mapping heuristic is effective in avoiding contention and creates schedulable mappings for the example applications.

Finally, we executed the examples on the SCC to verify the correctness of the mapping, the design of the scheduler, and the implementation of the run-time system. The execution also affirmed that all tasks can be executed successfully within the assumed timing constraints.

D. Framework

Figure 1 shows the flow of information between the individual tools in the proposed framework. The top layer is the application, which comprises the implementations of the individual tasks in C, and a description of the communication between tasks in a PRELUDE source file (`.plu`). This description is translated by PRELUDE[1] [3], which generates C code that implements the communication between tasks. We kept the general communication mechanism, but modified PRELUDE to leave the allocation of communication buffers to later stages and to access them via functions instead of direct reads and writes. Furthermore, PRELUDE writes a description of the tasks' properties, their dependencies, and the buffers required for communication to a `.txt` file. This task set description is mapped to the SCC by INTERLUDE. INTERLUDE emits a description of the task set like PRELUDE, with additional information about the mapping of tasks to cores. This description is then passed on to the schedulability analysis. In case the generated mapping is unschedulable, the description can be edited to pre-map tasks to cores and avoid undesirable configurations. INTERLUDE also generates C code describing the mapping of tasks to cores, their dependencies, and the locations of communication buffers. This code is compiled together with a scheduler skeleton and libraries of the bare-metal framework BAREMICHAEL with extensions by Scheller [11]. The resulting binaries can be executed directly on the SCC.

[1] PRELUDE is available at http://www.lifl.fr/~forget/prelude.html

E. Outline

The paper is organized as follows: Section II describes related work with regard to mapping techniques and execution environments for real-time systems on many-core processors. Section III describes the system model used throughout this paper. The run-time part of the framework is detailed in Section IV, while Section V describes the mapping heuristic and schedulability analysis. The experimental evaluation of the framework is presented in Section VI. Finally, Section VII concludes the paper and provides an outlook on future work.

II. Related work

A. Mapping Techniques

For partitioned scheduling, it is necessary to find a reasonable mapping of tasks to cores. Finding such a mapping corresponds to a *bin-packing* problem [12], which is NP-hard. Therefore, several heuristics have been developed to find approximate solutions.

Lombardi et al. developed a technique to construct robust partitioned schedules for non-preemptive task sets that takes into account task precedences [13]. The task model considered by that paper is very similar to the task model we envision for the SCC. While the technique scales to tens of tasks, it most likely does not scale beyond that number due to the NP-hardness of the problem.

The SynDEx framework[2] maps hard real-time tasks to multiprocessor platforms while taking into account communication requirements. However, that framework assumes *strict periodicity*, i.e., that the start times of jobs of a task with period T are exactly T time units apart. This entails that communicating tasks execute at the same rate [14]. In contrast, our framework allows communication between tasks of arbitrary rates. Due to the focus on strict periodicity, the schedulability results of SynDEx [14] and the mapping heuristics [15] are interesting with regard to their general approach, but cannot be directly reused for our framework.

The work by Carle et al. [16] focuses on mapping synchronous data-flow models to time-triggered architectures. While the execution model assumed in that paper is different from the one presented in this paper, the respective task set transformations could be used to extend the framework presented in this paper to more flexible task sets.

Fisher and Baruah developed a heuristic for partitioned non-preemptive EDF scheduling [10]. For randomly generated task sets, the heuristic can schedule more than 80% of the task sets for system loads up to 3.6 on a platform with 4 processor cores. However, the heuristic does not take into account dependencies, and it remains to be seen how the heuristic performs for realistic task sets.

Buttazzo et al. propose a partitioning heuristic that takes into account task dependencies [17]. Again, the general approach is interesting, but not directly applicable to the envisioned task model. In particular, that approach assumes full preemptability of tasks, while we assume non-preemptible scheduling.

[2]SynDEx is available at http://www.syndex.org

B. Real-Time Execution Environments

Due to the novelty of many-core architectures, support for the execution of real-time tasks on such architectures is limited. The only real-time operating system targeting many-core processors we are aware of is OpenComRTOS [18].

OpenComRTOS is a communication-oriented real-time operating system developed by Altreonic. It features a micro-kernel design in which tasks communicate through ports using a generic message format. OpenComRTOS on the SCC uses ring-buffers for the communication between the cores; since there may be multiple writers they need to use locking. In order to signal the receiver that he has received a message, OpenComRTOS uses inter-core interrupts. In contrast to OpenComRTOS, the framework presented in this paper targets a specific system model and therefore provides less flexibility. On the other hand, this enables our framework to reduce overheads and avoid mechanisms such as interrupts or locking that could introduce timing unpredictability.

The parMERASA project [19] proposes the Manycore Operating System for Safety-Critical systems architecture (MOSSCA). The idea is to mimic the ARINC653 partitioning scheme by allocating one ARINC partition or even process onto a single core of a many-core processor. MOSSCA would have a micro-kernel architecture that shares some similarities with our bare-metal approach: only very basic resource management is provided by MOSSCA. The MOSSCA idea is currently investigated using a many-core processor simulator, and will ported to the parMERASA many-core processor.

III. System Model

The system model considered in this paper comprises a set of dependent periodic tasks $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$, a set of dependencies $\mathcal{R}$, and communication requirements $\mathcal{C}$. Each task τ_i is characterized by its period T_i, its WCET C_i, its offset O_i, and its (relative) deadline D_i.

The dependencies $\mathcal{R}$ are described by a set of *extended precedence constraints*. Extended precedence constraints describe which jobs of a predecessor task must have finished before respective jobs of the successor tasks may start execution. As extended precedence constraints specify precedences at the job-level (instead of the task-level), they can model dependencies between tasks at arbitrary rates. To denote that job l of task τ_k depends on job j of task τ_i, we write $\tau_{i.j} \rightarrow \tau_{k.l}$. The dependency pattern repeats at the hyperperiod of the two tasks, i.e., the least common multiple of their periods. The description of precedences at the job level can be brought to the task level by assuming that all jobs are dependent. In this case, the notation can be simplified to $\tau_i \rightarrow \tau_k$.

The communication requirements $\mathcal{C}$ describe the data flow between tasks. In the context of this paper, we are interested in particular in the size of individual messages between tasks, and the number of messages that must be buffered to ensure correct communication. To denote that data flows from output variable o of job $\tau_{i.j}$ to input variable v of $\tau_{k.l}$, we write $\tau_{i.j}.o \rightarrow \tau_{k.l}.v$.

Table I
EXAMPLE TASK SET

$\mathcal{S}$	T_i	O_i	C_i	D_i	$\mathcal{R}$	$\mathcal{C}$	Core
τ_1	2	0	1	2	$\tau_{2.0} \rightarrow \tau_{1.1}$	$\tau_{2.0}.o \rightarrow \tau_{1.1}.i$	0
τ_2	2	0	1	2	$\tau_{1.0} \rightarrow \tau_{2.0}$	$\tau_{1.0}.o \rightarrow \tau_{2.0}.i$	1
τ_3	4	0	2	4	$\tau_{1.0} \rightarrow \tau_{3.0}$	$\tau_{1.0}.p \rightarrow \tau_{3.0}.i$	2

Example 1. Table I shows an example task set with extended precedences and communication requirements. The set of tasks $\mathcal{S}$ comprises the three tasks τ_1, τ_2, and τ_3. Job 0 of both τ_2 and τ_3 depends on job 0 of τ_1; job 1 of τ_1 depends on job 0 of τ_2. While there is a cyclic dependency between τ_1 and τ_2 on the task level, this cycle is broken by the use of extended precedence constraints. The communication requirements mirror the precedence constraints. However, they detail that τ_2 consumes output variable o of task τ_1, while τ_3 consumes output variable p. This example will be used throughout Section IV to detail our approaches to scheduling and communication.

IV. RUN-TIME ENVIRONMENT

This section describes the bare-metal run-time environment developed for partitioned non-preemptive EDF scheduling of communicating periodic task sets. First, we briefly describe the specifics of the SCC platform and the BAREMICHAEL bare-metal framework. We then detail the tick-based scheduler implementation, which relies on the notion of a temporal gap. We also describe the communication between tasks through the message passing mechanisms of the SCC.

A. Intel SCC

The SCC is a research chip that was created to ease research on many-core architectures [6], [7]. It comprises 24 dual-core tiles organized in a 6×4 matrix, connected via an on-chip network. The structure of the SCC is shown in Figure 2, which also shows the four memory controllers of the SCC (labeled MC) and its connection to the outside world via an FPGA.

Each tile contains two cores, a router, and some memory to facilitate message passing. The cores are derived from relatively simple early Pentium processors, and contain a 16 KB instruction cache, a 16 KB level 1 data cache, and a 256 KB level 2 data cache. The routers connect the cores to the network; the routing policy is XY-routing, i.e., network packets travel first along the X-axis and then along the Y-axis to towards their destination. The memory area for message passing is 16 KB per tile and usually referred to as message passing buffer (MPB). As processors can read from and write to all MPBs, the MPBs form a distributed shared on-chip memory. The cost for accessing a remote MPB depends on the distance between the processor's tile and the tile containing the MPB. Note that due to a hardware bug even local accesses to the MPB must go through the tile's router [7].

B. Bare-Metal Library

For programming the SCC, we use the BAREMICHAEL bare-metal programming framework [20], with extensions by

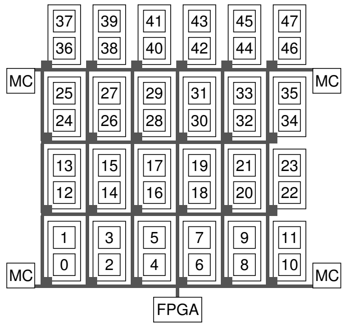

Figure 2. Intel SCC overview

Scheller [11]. The minimalist nature of the framework avoids overheads that come along with more complete frameworks. Also, the framework leaves us full control over the operation of the SCC, which is key for achieving predictability.

The processor cores of the SCC do not boot at the same time; their local clocks may have considerable offsets with regard to each other. Furthermore, in contrast to more usual multicore processors, access to a global clock is expensive on the SCC and always subject to delays caused by the network. Consequently, it is impossible to perfectly synchronize the core-local notions of time. However, the local clock signals are derived from a global clock signal. Therefore, there is no clock drift between the local clocks, which eliminates the need for synchronizing the core-local notions of time during execution. The bare-metal library provides means to synchronize the core-local clock notions to a precision of 4 μs.

C. Scheduler Implementation on the SCC

The scheduler implementation has been developed in the context of the SCHEDMCORE[3] scheduling framework. While the scheduler for the SCC is independent from SCHEDMCORE's scheduler, they share some ideas and bits of code. As in the current version of SCHEDMCORE, scheduling on the SCC is tick-based. This means that scheduling decisions are taken only at discrete instants rather than potentially every clock cycle. Using ticks as logical time of the schedulers simplifies the establishment of a consistent view of the ordering of tasks.

Figure 3 shows an example of tick-based scheduling for the three tasks τ_1, τ_2, and τ_3 with task properties as shown in Table I. At tick 0 all tasks check if their dependencies are met; this is the case for τ_1, but not for τ_2 and τ_3. Task τ_1 finishes execution at some point between tick 0 and tick 1. At tick 1, τ_2 and τ_3 find that job 0 of τ_1 has finished execution and start execution. At tick 2, job 1 of τ_1 is released. As job 0 of τ_2 has completed, it starts execution.

On the SCC, the scheduler implementation has to take into account that the local clocks are not perfectly synchronized, and that communication between cores can take a notable amount of time. To ensure that tasks on all cores have a consistent view of the state of other tasks, our scheduler (a) does not declare jobs

[3]SCHEDMCORE is available at http://sites.onera.fr/schedmcore/

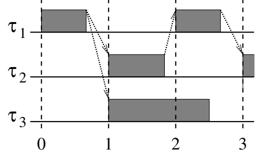

Figure 3. Tick-based scheduling

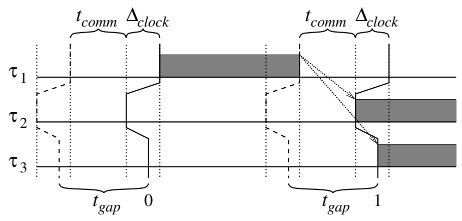

Figure 4. Gap before scheduling ticks 0 and 1

terminated before their WCET has expired, and (b) imposes a gap between the termination of a job and the beginning of the next tick. By doing so, the scheduler ensures that dependent tasks see the termination of predecessor jobs at the same tick. This approach is inspired by the concept of *sparse time* as advocated by Kopetz [21]. In particular, we use intervals of silence to ensure a consistent view of events (the termination of jobs), and start actions only at predefined points in time, the ticks.

With a maximum clock offset of Δ_{clock} and a maximum communication delay t_{comm}, this gap t_{gap} has to fulfill

$$t_{gap} \geq \Delta_{clock} + t_{comm}$$

As mentioned in Section IV-B, we can synchronize the local notions of time, such that $\Delta_{clock} = 4\ \mu$s. As the local clocks do not drift with regard to each other, Δ_{clock} remains constant during execution, and it is sufficient to perform the clock synchronization once during start-up. The evaluation in Section VI includes an empirical evaluation of t_{comm}.

Figure 4 visualizes the operation of the scheduler for ticks 0 and 1 of the example 1. The skewed, roughly vertical lines signify the local notion of time and deviate by at most Δ_{clock} between the cores. The solid lines among these signify the actual tick, while the dashed ones show the gap before the tick. To an external observer, ticks happen at different times at different cores. Task τ_1 executes on the "latest" core, while τ_2 runs on the "earliest" core and τ_3 at a core somewhere in between. At tick 0, only the dependencies for τ_1 are fulfilled, and it starts execution. When τ_1 terminates, it sends a message to τ_2 and τ_3. To ensure that both τ_2 and τ_3 receive this message in time (at latest when their local tick 1 arrives) it is necessary that τ_1 sends this message at latest $\Delta_{clock} + t_{comm}$ before its local tick 1. With the described scheme, it is not necessary to use synchronization mechanisms such a semaphores; the progress of time ensures the consistency of communication and the correct handling of precedence constraints.

D. Communication

Our framework uses the message passing buffers (MPBs) for the communication between tasks and the handling of precedence constraints. There are two basic policies when using the MPBs for communication between tasks: pushing data to the receiver's MPB, or letting the receiver pull data from the sender's MPB. Pushing data has the advantage that remote writes require only unidirectional communication over the network, while remote reads require a request and a response. Pulling data is beneficial for broadcasts, because the overhead from remote communication is spread out among the receivers.

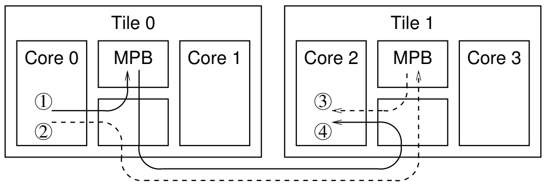

Figure 5. Communication example

The core-local schedulers must exchange information about which jobs of which tasks have finished in order to correctly handle dependencies. In essence, this information are the job counters of predecessor tasks. In order to minimize the network traffic, this information is always pushed, i.e., the job counters are always allocated in the MPB of the tile that contains the successor task. As an MPB is large enough to hold the job counters of around 4000 distinct predecessors, we find this approach suitable even for very large realistic systems.

Communication buffers may be allocated in an MPB or in external memory. We assume that all communication buffers are allocated either in the MPB of successor's or the predecessor's tile. By the default, communication buffers are allocated in the MPB of the successor's tile and data is pushed. If this is not possible, the buffer is allocated at the predecessor and data is pulled. For cases where allocation in an MPB is not possible, the mapping tool places some communication buffers in external memory. As such a case did not occur in the case studies we have evaluated so far, we chose not to model this case in the cost functions.

Figure 5 shows the communication between tasks τ_1 and τ_3 of the task set shown in Table I at tick 1. For the sake of clarity, the communication between tasks τ_1 and τ_2 is omitted. The example assumes that τ_1 executes on core 0 and τ_3 on core 2. The respective communication buffer is located in the MPB of tile 0, i.e., data is pulled. The continuous arrows refer to the flow of data according to the communication requirements, while dashed lines refer to communication for the handling of precedences. In step ①, task τ_1 writes to the communication buffer according to the protocol generated by PRELUDE. When τ_1 has finished, the scheduler of core 0 notifies τ_3 by writing to the MPB of tile 1 in step ②. When the scheduler checks the dependencies of τ_3 at the next tick, it reads from its own MPB in step ③. Finally, in step ④, task τ_3 pulls data from the communication buffer located in the MPB of tile 0.

In case the placement of buffers in external memory turns out to be an issue, future work could consider communication mechanisms that do not allocate the communication buffers directly in the MPBs.

V. Mapping Algorithm

The mapping algorithm takes as input the task set $\mathcal{S}$, the precedence constraints $\mathcal{R}$, and the communication requirements $\mathcal{C}$. The mapping algorithm's output is a mapping of tasks to cores, $\mathcal{M} = \{(\tau, c) \mid \tau \in \mathcal{S},\ c \in \mathbb{N}\}$, and a mapping of communication buffers to locations, $\mathcal{B} = \{(b, l) \mid b \in \mathcal{C},\ l \in \mathit{Loc}\}$. A location may be an address in a tile's MPB (such data is pushed by default, or pulled if that is not possible) or an address in external RAM.

This section presents model for these costs for the SCC, based on the properties of the hardware and the proposed scheduler implementation. Furthermore, this section presents a heuristic to generate a schedulable partitioning that minimizes the communication costs.

A. Cost Functions

The costs considered by the heuristic comprise three functions: *notification costs*, which consider the costs for dependency handling, the *contention*, which models potential contention for accesses to the MPBs, and the *overall traffic* on the network.

For the following, we define the set of successors of a task, $\mathit{Succs}(\tau)$, and the set of predecessors, $\mathit{Preds}(\tau)$, as follows:

$$\mathit{Succs}(\tau) := \{\tau' \in \mathcal{S} \mid \tau \rightarrow \tau'\}$$
$$\mathit{Preds}(\tau) := \{\tau' \in \mathcal{S} \mid \tau' \rightarrow \tau\}$$

1) Notification Costs: The communication delay depends on the time it takes to transmit the precedence information to the successors. Thus t_{comm} can be split into a local part t_{notif} and a network part t_{mesh}.

$$t_{comm} = t_{notif} + t_{mesh}$$

The notification costs t_{notif} reflect the time required for handing over the appropriate messages to the network, while t_{mesh} takes into account the worst-case traversal time of messages over the network.

In principle, t_{mesh} depends on the distance between the sender and the receiver in the network. However, our experiments could not establish a clear relationship between the distance in the network and the worst-case traversal time. Therefore, our cost model assumes that t_{mesh} is constant; as the experiments in Section VI show, this assumption is valid if the contention on the network remains sufficiently low. In this case, we can assume that t_{mesh} is lower than 10 μs.

The notification information is written only by the predecessor and is the same for all successors. Therefore, it can be shared by successors that are mapped to the same tile without inferring any costs. Consequently, t_{notif} depends on the number of different tiles that successors are mapped to, n_{notif}. Let $\pi(c)$ denote the set of tasks that are mapped to core c. Furthermore, let $\mathit{tile}(\tau)$ be the tile that τ is mapped to. For a given core c, the number of tiles to be considered for notification is given by

$$n_{notif}(c) := \max_{\tau \in \pi(c)} |\{\mathit{tile}(\tau') \mid \tau' \in \mathit{Succs}(\tau)\}|$$

With t_{send} denoting the time for putting a notification message onto the network, t_{notif} is defined as

$$t_{notif}(c) := n_{notif}(c) \times t_{send}$$

In order to ensure that all successors are notified during a tick gap, we must have that

$$t_{gap} \geq \Delta_{clock} + t_{mesh} + \max(t_{notif}(c))$$

In order to keep the overhead for the tick gap low, the primary objective function of the heuristic is to minimize $\max(n_{notif}(c))$.

2) Contention: While no single core can saturate the SCC's network, many cores accessing a single endpoint of the network could be problematic. In order to avoid such a bottleneck, the heuristic attempts to group tasks that read or write to the same MPB onto as few cores as possible.

The number of cores that can access the MPB of tile t depends on the predecessors, which update notifications or push data to communication buffers, and the successors, which may pull data from communication buffers. Let $\pi(t)$ be the set of tasks mapped to a tile t. The number of cores that can access the MPB of tile t for notifications or for writing to communication buffers is given by:

$$n_{cont}(t) := |\{\mathit{core}(\tau') \mid \tau \in \pi(t), \tau' \in \mathit{Preds}(\tau) \cup \mathit{Succs}(\tau)\}|$$

In order to minimize contention, the secondary objective is to minimize $\max(n_{cont}(t))$.

3) Overall Traffic: A third metric for the mapping is the overall amount of traffic. We define the distance between two tiles a and b at coordinates (a_x, a_y) and (b_x, b_y) as follows:

$$\mathit{dist}(a, b) := 1 + |a_x - b_x| + |a_y - b_y|$$

This corresponds to the number of routers that messages between two tiles have to pass.

The traffic model assumes that each task writes to all its successors once per period. The cost of each communication is the square of the distance between tasks:

$$\mathit{traffic} := \sum_{\tau_i} \sum_{\tau_k \in \mathit{Succs}(\tau_i)} \frac{\mathit{dist}(\mathit{tile}(\tau_i), \mathit{tile}(\tau_k))^2}{T_i}$$

The rationale behind this metric is twofold. On the one hand, reducing the overall traffic reduces the power consumption of the network. On the other hand, the notification and contention costs are the same for a large number of mappings and do not "guide" the heuristic towards a good solution. Using the overall traffic as tertiary objective function forces the heuristic to consider more configurations. As a consequence, the heuristic is more likely to gravitate towards a solution that improves the other cost functions. Using the square of the distance as cost penalizes long paths and provides a stronger incentive to cluster communicating tasks in the same region of the chip than using just the distance. During our experiments, we found that this tends to result in better overall solutions.

B. Heuristic

In order to be able to trade off computational effort and optimization of the cost functions, we implemented different levels of optimization. The "first-fit" level only considers schedulability and adds tasks to the first core on which they are schedulable. The "greedy" level adds tasks to the core that optimizes the cost functions according to the comparison function described below. The "move" and "exchange" levels try to further optimize the result of the greedy level by moving and/or exchanging tasks between cores. As the first-fit and the greedy approaches do not revert earlier decisions, the order in which tasks are passed to the algorithm is important. We propose an order that reflects the communication between tasks.

For the sake of efficiency and simplicity, the mapping heuristic uses simple dependencies instead of extended precedence constraints. As we do not allow migration, the heuristic maps tasks rather than jobs to cores, which makes it natural to use dependencies on the task level. As extended precedences model dependencies between individual jobs, tasks can appear to have cyclic dependencies, which make topological sorting at the task level impossible. In order to break these cycles, we propose an approximation of topological sorting.

We approximate topological sorting by sorting task τ_i before task τ_k if τ_k (transitively) depends on τ_i, but τ_i does not (transitively) depend on τ_k. If such an order cannot be established, tasks are ordered in decreasing order of their number of successor tasks. For the task set given in Table I, tasks τ_1 and τ_2 depend on each other and are therefore considered unordered with regard to their dependencies. Due to its greater number of successor tasks, τ_1 is sorted before τ_2.

To determine if a task can be added to the core, the heuristic uses the schedulability conditions by Fisher and Baruah [10]. As these conditions do not take into account dependencies, they only approximate schedulability and might consider task sets schedulable that are in fact unschedulable. In order to increase the likeliness of generating a schedulable partitioning, we additionally require that the load factor remains below a certain limit:

$$\sum_{\tau_i \in \pi(core(\tau))} \frac{C_i}{\min(D_i, T_i)} \leq |\pi(core(\tau))| \times (2^{1/|\pi(core(\tau))|} - 1)$$

This condition—akin to the classic schedulability condition for rate-monotonic scheduling—is not a schedulability test. However, we found during our experiments that it helps the heuristic in generating valid partitionings. As none of the above conditions is an accurate schedulability test for non-preemptive partitioned scheduling of a dependent task set, the schedulability of the partitioning has to be validated by a separate schedulability analysis.

If a task τ can be added to more than one core, the heuristic uses the relation $\sqsubset$ to order potential mappings. With n_a being the notification cost of a mapping a, c_a the contention, t_a the overall traffic, and l_a the load factor, defined as $\sum_{\tau_i \in \pi(core(\tau))} \frac{C_i}{\min(D_i, T_i)}$, we define the comparison of two mappings a and b as

$$\begin{aligned} a \sqsubset b := & (n_a < n_b) \vee (n_a = n_b \wedge c_a < c_b) \\ & \vee (n_a = n_b \wedge c_a = c_b \wedge t_a < t_b) \\ & \vee (n_a = n_b \wedge c_a = c_b \wedge t_a = t_n \wedge l_a < l_b) \end{aligned}$$

More intuitively, the heuristic picks the core that minimizes in decreasing order of importance (a) notification costs, (b) contention, (c) overall traffic, and (d) the load factors. Using the load factors as tie breaker balances the load across processor cores. This leaves more freedom to the allocation of tasks as the greedy heuristic progresses, and is also more likely to result in a schedulable partitioning.

The move and exchange optimization levels try to further optimize the result of the greedy approach. The move approach iterates over all tasks and moves each task to the core that results in the best solution according to the above comparison function. This process is iterated until a fixed point is reached. The exchange approach first iterates over all tasks and optimizes the solution by moving tasks, and then iterates over all pairs of tasks and exchanges them if this results in a better solution. Again, this process is repeated until a fixed point is reached. While the effort for optimizing the partitioning by moving tasks is moderate, exchanging tasks is considerably more expensive. These optimizations can be helpful in cases where the default greedy heuristic generates a result with too high notification costs or contention to be useful. Also, these optimizations provide a simple way to generate alternative partitionings in case the partitioning generated by the greedy heuristic is not schedulable.

Task precedences are not taken into account by the schedulability condition by Fisher and Baruah [10]. Therefore, its results are not safe for the task sets generated by PRELUDE. We couple the mapping heuristic with a schedulability analysis, to verify the validity of the partitioning. The schedulability analysis does not take into account the tick gap; we assume that the task WCETs are computed such that they include the tick gap. This can be done by adding the time for the tick gap to the WCET in cycles or physical time, before transforming it to a WCET in terms of ticks.

C. Partial Schedulability Analysis

The schedulability analysis uses an exact approach that simulates the execution of the task set until a violation of the timing constraints is detected or an already explored state is reached [22]. To handle partitioned non-preemptive scheduling, we SCHEDMCORE extended in two regards:

- For managing non-preemptive tasks, we ensure that once a task is started, it will not be stopped until termination.
- Instead of a unique global queue, we use one queue per processor. The local queue is sorted according to the policy and the global dependencies.

The schedulability analysis (i.e., the simulation of the task set), is performed by a custom C program. While the "brute force" approach of simulating execution might seem expensive, it works well in practice, as the results in Section VI indicate.

The mapping is not guaranteed to find an optimal solution with regard to schedulability or minimizing the cost functions detailed above. Determining if there exists a mapping where all tasks can satisfy their timing constraints is already an NP-hard problem. Additionally trying to optimize the communication between tasks does not simplify the problem. However, we are interested in finding *some reasonable* solution rather than an optimum. As shown in Section VI, the heuristic finds mappings for reasonably large, realistic benchmarks.

VI. EVALUATION

The evaluation presented in this section comprises two parts. The first part experimentally determines the tick gap that is necessary to ensure the correctness of our scheduling approach. The second part applies the mapping heuristic to example applications and evaluates its effectiveness. Also, the example applications are executed on the SCC to verify the correctness of the overall system. For the measurements described in this section, the clock frequency of all cores is 533 MHz, and the on-chip network as well at the memory controllers operate at 800 MHz. The measurements and the execution of task sets rely solely on a minimal C library and the BAREMICHAEL framework for processor initialization and accessing the hardware.

A. Tick Gap

The tick gap is vital for the correctness of our scheduling approach. While the core-local schedulers can detect if the tasks they schedule overrun their time budget, they cannot detect if the tick gap is too small for reliable communication between cores. Therefore, is is necessary to derive a reliable estimate for t_{gap}.

1) Clock precision: The first component of the tick gap is the maximum offset between the cores' clocks, Δ_{clock}. As the individual cores do not boot at the same time, the scheduler starts with a barrier to synchronize execution on the individual cores. In order to estimate Δ_{clock}, we compare the readings of the global clock after this barrier. As simultaneous reads would lead to contention that could distort the measurements, each core waits for $i \times 10$ μs (i being the core number) before reading. As the global clock is only accessed in the synchronization barrier and this measurement, avoiding contention on the global clock is not necessary during regular execution. In accordance with earlier results [11], we observed a maximum clock offset of 4 μs.

2) Costs for sending: The second component of the tick gap is the time required for notifications, t_{notif}, which in turn depends on the worst-case time for putting a message onto the network, t_{send}. Ideally, we would have a formally proved worst-case bound for t_{send}. Unfortunately, no such bound is available in the literature, and deriving a formal model of the SCC goes beyond the scope of this paper.

In the experiments, a test node k writes repeatedly to the MPBs of all other tiles and measures the time required for each write. During these measurements, the first n cores (apart from core k) flood the network with accesses to a shared resource, i.e., reads or writes to core k's MPB or the external RAM. The measurements are repeated for each core, and for increasing numbers of cores that produce noise. We also explored other schemes than using cores 0 to n-1 to generate noise, which however did not change the outcome of the measurements.

The measurement results in Figure 6 show that in most cases, t_{send} stays below 10 μs for up to around 40 cores that produce noise. There are two exceptions to this: First, t_{send} quickly increases to several hundred milliseconds if 18 or more cores read from the test core's MPB. Further experiments showed that high contention due to reads from a tile's MPB can in fact lead to t_{send} becoming arbitrarily large. The heuristic takes this into account by minimizing n_{cont}.

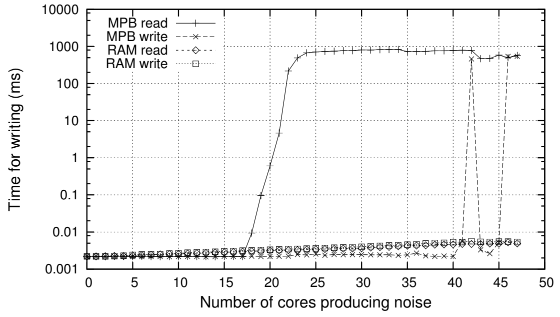

Figure 6. Measured t_{send} for increasing number of interfering cores.

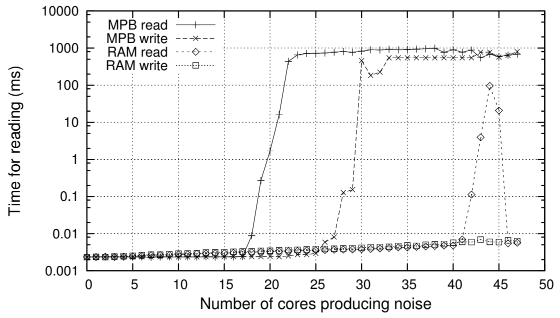

Figure 7. Round-trip times for increasing number of interfering cores.

The second exception, not visible in Figure 6, are rare outliers than can occur when monitoring the application's output. The implementation of `printf` in the BAREMICHAEL library dumps its output to the SCC's external memory, from where the host PC fetches it via the FPGA shown at the bottom of Figure 2. This can cause interferences in the order of 100 μs. In the following, we assume that this instrusive communication mechanism is used during development only and avoided when deploying the system.

According to the results presented above, we can assume that t_{send} is 10 μs, if less than 18 cores access the same MPB or external memory via the same memory controller. For accesses to the MPBs, the results from the mapping heuristic indicate if this precondition holds. For accesses to external RAM, the precondition has to be ensured by the application designer. As accesses to private memory are evenly distributed to the external memory controllers, this concerns only the use of external memory for (implicit) communication between tasks.

3) Traversal time: The worst-case traversal time of packets in the network, t_{mesh}, is the third component of the tick gap. As a direct measurement would be influenced by the clock offset between the sending and receiving core, we estimate t_{mesh} through the time required to read from a remote MPB. Reads require one packet for the request and one packet for the response. Therefore, we in fact measure the round-trip time, which gives an upper bound for the traversal time. The experiments use the same setup as for the evaluation of t_{send} with the only difference that the test core reads from all other MPBs instead of writing to them.

The results for the round-trip time are similar to the results for t_{send}. Again, 10 μs are a suitable upper bound when avoiding

Table II
FAS TASK PROPERTIES AND MAPPING OF TASKS TO CORES BY GREEDY HEURISTIC

Task	Name	Period	Offset	WCET	Deadline	Predecessors	Core
τ_1	GNC_DS	1000	0	300	1000	$\tau_{16.0} \rightarrow \tau_{1.0}$	4
τ_2	tm	10000	0	10	10000	$\tau_{12.0} \rightarrow \tau_{2.0}$	2
τ_3	str	10000	0	10	10000		0
τ_4	PDE	100	0	30	100	$\tau_{17.0} \rightarrow \tau_{4.0}, \tau_{1.0} \rightarrow \tau_{4.10}$	5
τ_5	Gyro_Acq	100	0	30	100	$\tau_{6.0} \rightarrow \tau_{5.0}, \tau_{12.0} \rightarrow \tau_{5.100}$	5
τ_6	gyro	100	0	10	100		1
τ_7	gps	1000	0	10	1000		2
τ_8	gnc	1000	0	10	300	$\tau_{16.0} \rightarrow \tau_{8.0}$	2
τ_9	Str_Acq	10000	0	30	10000	$\tau_{3.0} \rightarrow \tau_{9.0}, \tau_{12.0} \rightarrow \tau_{9.1}$	3
τ_{10}	pde	100	0	10	100	$\tau_{4.0} \rightarrow \tau_{10.0}$	2
τ_{11}	GPS_Acq	1000	0	30	1000	$\tau_{7.0} \rightarrow \tau_{11.0}, \tau_{12.0} \rightarrow \tau_{11.10}$	4
τ_{12}	TM_TC	10000	0	1000	10000	$\tau_{17.0} \rightarrow \tau_{12.0}, \tau_{13.0} \rightarrow \tau_{12.0}$	3
τ_{13}	tc	10000	0	10	10000		3
τ_{14}	PWS	1000	500	30	1000	$\tau_{1.0} \rightarrow \tau_{14.0}$	4
τ_{15}	SGS	1000	0	30	1000	$\tau_{1.0} \rightarrow \tau_{15.0}$	4
τ_{16}	GNC_US	1000	0	210	1000	$\tau_{9.0} \rightarrow \tau_{16.0}, \tau_{5.0} \rightarrow \tau_{16.0}, \tau_{17.0} \rightarrow \tau_{16.0}, \tau_{11.0} \rightarrow \tau_{16.0}$	4
τ_{17}	FDIR	100	0	15	100	$\tau_{16.0} \rightarrow \tau_{17.10}, \tau_{9.0} \rightarrow \tau_{17.0}, \tau_{5.0} \rightarrow \tau_{17.0}, \tau_{11.0} \rightarrow \tau_{17.0}$	5
τ_{18}	sgs	1000	0	10	1000	$\tau_{15.0} \rightarrow \tau_{18.0}$	4
τ_{19}	pws	1000	500	10	1000	$\tau_{14.0} \rightarrow \tau_{19.0}$	4

Table III
CASE STUDY SUMMARY

Case Study	Tasks	Deps.	Buffers	Buffer Size	Utilization
FAS	19	26	26	616 B	1.696
FAS_complete	236	331	332	6556 B	10.340
Asm	14	20	20	304 B	1.595
Asm_complete	375	420	514	3260 B	5.063

contention. The similarity between reads and writes indicates that the observed times are dominated by contention effects rather than the time to transmit a packet over the network. Under the preconditions outlined above, we can compute t_{gap} in μs according to

$$t_{gap} \geq \Delta_{clock} + t_{mesh} + \max(t_{notif}(c))$$
$$t_{gap} \geq 4 + 10 + \max(n_{notif}(c)) \times 10$$

B. Case Studies

The first case study we evaluate is a simplified version of the Flight Application Software (FAS) of the Automated Transfer Vehicle designed by EADS Astrium Space Transportation for resupplying the International Space Station. The properties of this case study's task set are shown in Table II. The second case study is a more complete version of FAS. Due to its size of more than 200 tasks, the detailed task properties are omitted here. The basic properties of this and the other case studies are shown in Table III. Its size makes FAS_complete an excellent test case for the efficiency of the mapping algorithm and schedulability analysis. The third and fourth case studies, Asm and Asm_complete, are derived from a sub-system of a flight control system that is responsible for handling inputs by the pilot. Again Asm refers to a simplified version, while Asm_complete is a more complete variant. Like for FAS_complete, the size of Asm_complete makes it a test case for the efficiency of our framework.

Table IV subsumes the results of the mapping heuristic for the case studies. The results show that all optimization levels are effective in keeping n_{notif} and consequently t_{gap} reasonably low. The tick gap to be observed is in all cases less than 54 μs, which would correspond to an overhead of about five percent with a tick granularity of 1 ms. The greedy approach results in higher contention than the first-fit approach. This can be explained by the load balancing in the greedy heuristic, which spreads the tasks among a larger number of cores. The move and exchange optimization levels lead to the best results in terms of notification costs, contention, and overall traffic. Not all generated partitionings are schedulable. However, for each test case at least on level of optimization results in a schedulable partitioning. While the first-fit and greedy optimization levels can map all task sets within a few seconds (measured on an Intel Core i7 processor at 2.2 GHz), the move and exchange optimizations are considerably slower for the large task sets. The schedulability analysis terminates within less than 10 seconds even for the large case studies.

We executed the schedulable partitionings shown in Table IV on the SCC with a tick size of 1 ms and a tick gap of 100 μs. As expected, the traces from these executions matched traces generated on a general-purpose machine, meaning that the execution on the SCC was successful.

VII. CONCLUSIONS AND OUTLOOK

In this paper we have presented an end-to-end framework for the development of multi-periodic real-time systems targeting a many-core architecture. This framework allows the designer to formally specify for the system and to generate automatically a valid static mapping of tasks and communication buffers to the Intel Single-chip Cloud Computer (SCC). This mapping can then be executed using a real-time bare-metal run-time environment.

In the future, we will extend the scheduling policies on the core to support non-preemptive fixed-priority and off-line scheduling. The mapping algorithm will be extended to take into account schedulability under these policies. Also, we will explore non-greedy partitioning heuristics.

Table IV
MAPPING HEURISTIC RESULTS

Case Study	Heuristic	n_{notif}	n_{cont}	*traffic*	t_{gap} (us)	Schedulable	Cores	Time Heuristic (s)	Time Analysis (s)
FAS	First-Fit	2	4	0.187	34	no	4	0.04	0.00
	Greedy	2	5	0.229	34	yes	6	0.11	0.32
	Move	2	4	0.176	34	no	4	0.20	0.00
	Exchange	2	4	0.146	34	no	4	0.19	0.00
FAS_complete	First-Fit	3	14	2.431	44	no	14	1.35	0.19
	Greedy	3	18	3.716	44	no	48	1.95	0.09
	Move	3	9	1.766	44	no	47	12.68	0.24
	Exchange	3	9	1.717	44	yes	47	69.11	9.76
Asm	First-Fit	2	3	1.730	34	yes	3	0.02	0.00
	Greedy	2	4	1.730	34	yes	4	0.05	0.00
	Move	2	4	1.730	34	yes	4	0.10	0.00
	Exchange	2	4	1.190	34	yes	4	0.13	0.00
Asm_complete	First-Fit	4	8	9.738	54	no	8	1.52	0.82
	Greedy	3	31	8.051	44	no	48	5.62	0.32
	Move	3	9	4.022	44	yes	16	23.18	6.87
	Exchange	3	8	3.004	44	no	18	610.58	1.13

A second perspective concerns the definition of detailed timing model of the many-core platform. This step is essential to prove the safety of the overall real-time system. Such a model would make it possible to derive provable bounds on the worst-case execution time of tasks and to formally define the costs for communication over the on-chip network.

Acknowledgements: This work has been funded by the RTRA STAE foundation within the TORRENTS[4] cluster. The authors would like to thank Intel for granting an access to the SCC as part of the MARC program. The authors would also like to thank Johannes Scheller for his extensions on the BAREMICHAEL framework.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[2] R. Wilhelm and J. Reineke, "Embedded systems: Many cores – many problems," in *Symposium on Industrial Embedded Systems (SIES'12)*, 2012.

[3] J. Forget, "A synchronous language for critical embedded systems with multiple real-time constraints," Ph.D. dissertation, Université de Toulouse - ISAE/ONERA, Toulouse, France, Nov. 2009.

[4] H. Zeng and M. D. Natale, "Schedulability analysis of periodic tasks implementing synchronous finite state machines," in *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. IEEE Computer Society, 2012, pp. 353–362.

[5] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *20th International Conference on Real-Time and Network Systems (RTNS'12)*, 2012, pp. 11–19.

[6] Intel Labs, "SCC external architecture specification (EAS)," Intel Corporation, Tech. Rep., May 2010.

[7] ——, "The SCC programmer's guide," Intel Corporation, Tech. Rep., January 2012.

[8] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Real-Time Systems Symposium, 2008*, 30 2008-dec. 3 2008, pp. 157 –169.

[9] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, july 2005, pp. 41 – 48.

[10] N. Fisher and S. Baruah, "The partitioned multiprocessor scheduling of non-preemptive sporadic task systems," in *14th International COnference on Real-Time and Network Systems*, 2006.

[11] J. Scheller, "Real-time operating systems for many-core platforms," Master's thesis, ISAE/ONERA, Toulouse, France, 2012.

[12] M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. Yao, "Resource constrained scheduling as generalized bin packing," *Journal of Combinatorial Theory*, vol. 21, pp. 257–298, 1976.

[13] M. Lombardi, M. Milano, and L. Benini, "Robust non-preemptive hard real-time scheduling for clustered multicore platforms," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, april 2009, pp. 803 –808.

[14] L. Cucu and Y. Sorel, "Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints," in *In Proceedings of the 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group, PlanSIG'04*, 2004.

[15] O. Kermia and Y. Sorel, "A Rapid Heuristic for Scheduling Non-Preemptive Dependent Periodic Tasks onto Multiprocessor," in *Proceedings of ISCA 20th International Conference on Parallel and Distributed Computing Systems, PDCS'07*, Las Vegas, Nevada, USA, 2007.

[16] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens, "From dataflow specification to multiprocessor partitioned time-triggered real-time implementation," INRIA, Research report RR-8109, Oct. 2012.

[17] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 2, pp. 302 –315, may 2011.

[18] B. Sputh, A. Lukin, and E. Verhulst, "Transparent Programming of Many/Multi Cores with OpenComRTOS Comparing Intel 48-core SCC and TI 8-core TMS320C6678," in *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, E. Noulard and S. Vernhes, Eds. Toulouse, France: ONERA, The French Aerospace Lab, Jul. 2012, pp. 52–58.

[19] F. Kluge, B. Triquet, C. Rochange, and T. Ungerer, "Operating systems for manycore processors from the perspective of safety-critical systems," in *Proceedings of 8th annual workshop on Operating Systems for Embedded Real-Time applications (OSPERT 2012)*, 2012.

[20] M. Ziwisky and D. Brylow, "BareMichael: A minimalistic bare-metal framework for the intel SCC," in *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, E. Noulard and S. Vernhes, Eds. Toulouse, France: ONERA, The French Aerospace Lab, Jul. 2012, pp. 66–71.

[21] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*. IEEE, 1992, pp. 460–467.

[22] M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti, "Multiprocessor schedulability analyser," in *Proceedings of the 26th ACM Symposium on Applied Computing (SAC'11)*, 2011.

[4]http://www.irit.fr/torrents/index.php