KTS: a real-time mapping algorithm for NoC-based many-cores

Audrey Queudet, Nadine Abdallah, Maryline Chetto

KTS: a real-time mapping algorithm for NoC-based many-cores

Audrey Queudet[1] · **Nadine Abdallah**[2] · **Maryline Chetto**[1]

Abstract Many-core architectures based on network-on-chip (NoC) are scalable and have the ability to meet the increasing performance requirements of complex concurrent applications (real-time video, communications, control, etc.). This paper addresses the mapping problem of hard real-time task sets on NoC-based many-core processors. Our main contribution is a novel static mapping scheme called *K-level task splitting (KTS)*. If a task cannot be allocated on a given core of the NoC because of a too high processing utilization ratio, it gets replicated so that its jobs execute on more than one core without migrating. Synchronization between task replicas could then be ensured by assigning offsets and virtual deadlines to them. KTS's advantage is that data migration is not required, thus involving no overheads due to migrations. The only requirement is that all core clocks be synchronized within the NoC. In this newly proposed algorithm, the schedulability of each task is determined based upon fundamental results relative to the feasibility analysis of asynchronous real-time task sets. The paper describes the principles of task splitting, our algorithm and its properties. We evaluate the efficiency of KTS, demonstrating that it is a good compromise between existing semi-partitioned schemes (with possible migrations) and fully partitioned approaches.

Keywords Many-core real-time systems · NoC-based architectures · Mapping algorithms · Real-time scheduling

✉ Audrey Queudet
audrey.queudet@univ-nantes.fr

Maryline Chetto
maryline.chetto@univ-nantes.fr

[1] IRCCyN UMR CNRS 6597, 44321 Nantes, France

[2] IUT Saida, Lebanese University, Sidon, Lebanon

1 Introduction

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified time interval. The correctness of the result of a real-time task is not only related to its logic correctness, but also to when the results occur [17]. Traditional classification of real-time systems stands for two classes to characterize the real-time requirement of such systems: *hard* and *soft*. In *hard* real-time systems, an overrun in response time can lead to potential loss of life and/or big financial damage. For *soft* systems, deadline overruns are tolerable, but not desired. And there are no catastrophic consequences of missing one or more deadlines.

Today multiprocessor and multicore architectures are invariably used. This is not the case of the so-called many-core processors, whereas the International Technology Roadmap for Semiconductors foresees that the number of processing elements that will be integrated into an system-on-chip (SoC) will be in the order of thousands by 2020 [15]. The migration from traditional architecture to many-core systems brings vast works to deal with. In order to obtain the maximum performance, programmers have to consider a mass of things such as mapping tasks onto cores, task scheduling and even the cache coherence.

Network-on-chip (NoC)-based many-core processors offer a promising support to deal with a number of issues, such as system performance and power consumption. With a large-scale integration of multiple cores on a single chip, they provide a promising solution for real-time applications. To the best of our knowledge, partitioning for NoC-based many-cores has not been studied for real-time applications. One study [8] has explored the partitioning on NoC-based many-core processors for critical applications but with fault tolerance requirements only. The overall performance of NoC-based many-cores with respect to real-time requirements mostly depends on the global intrachip communications. Indeed, in the most efficient real-time mapping approaches existing today, tasks must migrate between cores. Frequent task migration results in additional communication costs and cache misses and potentially very high overheads which might seriously degrade application performance in a NoC-based system. Consequently, a key challenge for NoC-based systems is either to manage task migrations in a bounded and predictable way or to reduce the overhead by restricting/eliminating task migrations. The use of additional communication links, circuit-switched routing or prioritized communication channels is among existing solutions at the hardware level to guarantee the required latency. A number of process migration mechanisms and policies have also been proposed (see [14] for a survey) in order to reduce state transfers and exhibit low-overhead migration.

The main objective and scope of this work is to study the mapping problem of a real-time task set on NoC-based many-core processors. In our research we propose an approach with *no task migrations*. Hence, the cache effectiveness is not decreased, and tasks' execution times are not increased because of the need for fetching again data from the main memory. The only requirement of the approach is that all processor clocks be synchronized. Once the task mapping is done, each processor independently applies its local scheduler. Our partitioning algorithm only provides a *feasible* mapping. It is then the role and responsibility of the application designer to select a given

real-time scheduler that will produce a valid schedule in which all the timing requirements of the task set will be met. Ideally, the designer should select an optimal one which, by definition, is able to schedule every feasible task set. Otherwise, he should couple the results of the *feasibility* analysis with a *schedulability* analysis.

The remainder of the paper is structured as follows. The next section reviews related research in both multicore mapping and NoC clock synchronization. The system model is defined in Sect. 3. Section 4 presents K-level task splitting (*KTS*), a new static mapping strategy with an offline task splitting phase. Section 5 provides some results of experiments. Finally, Sect. 6 concludes with a summary of the contributions and outlines some directions for future work.

2 Related work

2.1 Mapping approaches for multiprocessor systems

As our work is closely related to classical results on the partitioning of real-time tasks on multiprocessor architectures, we provide hereafter a summary of mapping approaches for multiprocessor systems.

Partitioning a task set amounts to a bin-packing problem (known to be NP hard [11]): how to place n objects of different sizes in m identical boxes? Some heuristics have been proposed in the literature in order to solve it in an acceptable computational time [12]. Heuristics as *First Fit (FF)*, *Best Fit (BF)*, *Worst Fit (WF)* or *Next Fit (NF)* imply a sequential mapping of tasks to processors after passing a schedulability test. A task sorting criterion is used to determine the order in which tasks are examined for mapping.

Hybrid mapping approaches extend partitioned ones allowing a subset of tasks to migrate. Among them, semi-partitioning is often outlined as the best compromise with most tasks statically partitioned onto cores. It offers improved load balancing and increased utilization compared to static partitioning. The concept was introduced by Anderson et al. [1] with the *EDF-fm* algorithm dedicated to soft real-time systems. At most $m-1$ tasks need to be able to migrate, and each such task migrates between two processors, across job boundaries only. Most recently, in 2014, the same authors introduced another EDF-based semi-partitioning strategy called EDF-os [2]. EDF-os minimizes tardiness by introducing some constraints on migrations: (i) the number of processors to which jobs of a migrating task can migrate to at most two and (ii) each processor can be assigned to only two migrating tasks.

In [6], Burns et al. introduced an EDF-based task splitting scheme called *EDF Split (DD)*. Each processor p is filled with tasks until no further task can be added due to unschedulability on the processor. Any non-allocated task τ_i is split into two subtasks. The first one is retained on processor p while guaranteeing schedulability. The second one is mapped to the processor $p+1$ with offset computed so as to avoid the two generated tasks to execute simultaneously. The authors show that *EDF Split (DD)* makes a small but significant improvement over fully partitioned approaches. Evaluation of our approach is provided via comparative benchmarks with *EDF Split (DD)*, one of the best semi-partitioned algorithms found in the literature.

2.2 Clock synchronization in NoCs

As our approach requires that all processor clocks be synchronized within the NoC, we report hereafter some works about clock synchronization with a focus on those suitable for real-time applications. There is indeed a vast literature on the subject [5]. Several techniques have been proposed in the literature to address the issue of distributing a high-speed, low-power, low-jitter clock across the whole chip [18]. The most commonly used today consists in trying to align the entire chip with a global clock with low skew, yet challenging to fully master [16]. Another popular method is to make use of asynchronous protocols that are free from clock–skew issues but are prone to metastability problems due to asynchronous clock domain crossings [13]. Among existing asynchronous NoCs, only *MANGO* [4] offers hard real-time guarantees. A middle option, particularly interesting for NoC-based systems, is also possible: the *globally asynchronous, locally synchronous (GALS)* method [7]. It relies on blocks that are synchronous but communicate asynchronously. The *GALS* scheme has the benefit of allowing cores to decouple from one master clock. Hence, it appears as the only one that supports the challenge of building high-scalable systems. A real-time NoC architecture with an efficient *GALS* implementation named *Argo* has been proposed in [9].

3 Models and terminology

3.1 System and task model

We consider a NoC architecture which is intended to hard real-time applications as presented in [9]. Figure 1 shows such a homogeneous NoC $\Pi = \{\pi_1, \ldots, \pi_m\}$ with m identical cores in using a 2-D-mesh topology.

In a real-time context, the mapping problem for NoC is to decide how to place a set of n asynchronous real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ onto the cores of a given network such that all timing constraints are met.

Each task τ_i is assumed to be periodic and characterized by a 4-tuple (ϕ_i, C_i, T_i, D_i) whose parameters are described hereafter:

- ϕ_i: the offset (i.e. the time of the first activation of τ_i relatively to the system initialization time),
- C_i: the worst-case execution time (WCET),
- T_i: the period or minimal inter-arrival time between two consecutive activations of τ_i,
- and D_i: the relative deadline (i.e. the time by which the current job of τ_i has to finish its execution relatively to its arrival time).

The utilization of task τ_i, denoted u_i is given by $\frac{C_i}{T_i}$. The normalized system utilization of task set τ is defined as $u_{sys} = \sum_{i=1}^{n} u_i/m$.

Any task τ_i generates an infinite number of jobs τ_{ij}, $j \geq 0$. Each job is assigned a release time $r_{ij} = \phi_i + jT_i$ and an absolute deadline $d_i = r_{ij} + D_i$. When all offsets equal to zero, τ is said to be *synchronous*; otherwise, it is *asynchronous*. Tasks are

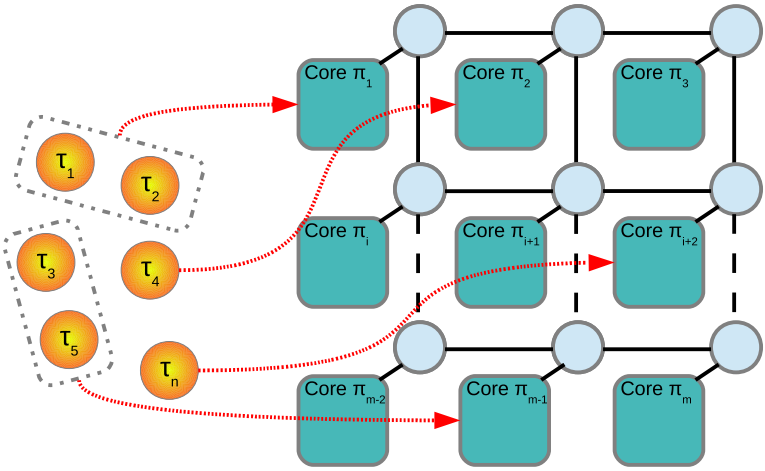

Fig. 1 Static mapping on a homogeneous NoC using a 2-D-mesh topology

Table 1 Notation used throughout the paper

Symbol	Description
n	Number of tasks
m	Number of processors
$\mathcal{T}$	Task set
Π	NoC core set
τ_i	Task i of the task set $\mathcal{T}$
π_j	Processor j of the core set Π
C_i	Worst-case execution time of task τ_i
D_i	Relative deadline of task τ_i
T_i	Period or minimal inter-arrival time of task τ_i
u_i	Utilization factor of task τ_i, $u_i = C_i/T_i$
u_{sys}	Normalized system utilization, $u_{sys} = \sum_{i=1}^{n} u_i/m$
Φ_j	Max offset of any tasks assigned to processor π_j
Φ	Max offset of any tasks in $\mathcal{T}$
$H(\mathcal{T})$	Hyperperiod of task set $\mathcal{T}$, $H(\mathcal{T}) = lcm(T_1, \ldots, T_n)$
K	Number of levels of task splitting

assumed to be preemptable and independent in the sense that there exists no kind of dependency (e.g. shared data resources, precedence ordering) of one job on another of the same or another task. The deadline of τ_i is less than or equal to its period (i.e. $\forall i$, $D_i \leq T_i$), also referred to as the *constrained-deadline* model. Note that in the special case where $\forall i$, $D_i = T_i$, tasks are said to have *implicit* deadlines.

For ease of reference, Table 1 provides a summary of notations used in the rest of the paper.

3.2 Feasibility analysis of asynchronous task sets

In our approach, the feasibility of the static mapping on the NoC is guaranteed by a core-by-core feasibility analysis, considering asynchronous independent real-time periodic tasks. Hence, this section summarizes previous results on feasibility analysis on uniprocessor platforms which has always been a central research issue in real-time scheduling.

Definition 1 A task set $\mathcal{T}$ is *feasible* if and only if there is some scheduling algorithm that will successfully schedule all job sequences that can be generated by $\mathcal{T}$.

The feasibility problem of asynchronous periodic task sets on a uniprocessor platform has been proven to be co-NP-complete in the strong sense [10]. Leung and Merril [10] proved that we have to analyse all deadlines in $[0, \Phi + 2H]$. In [3] Baruah et al. proposed an exact test for task sets with constrained deadlines. It is based on the processing demand within any time interval compared to the available processing capacity in that interval, as described in Lemma 1.

Lemma 1 [3] *A set of independent real-time tasks $\mathcal{T}$ is feasible on a single processor if and only if:*

1. $U = \sum_{\tau_i \in \mathcal{T}} \frac{C_i}{T_i} \leq 1$ *and*
2. $\forall 0 \leq t_1 < t_2 \leq \Phi + 2H, dbf\,(t_1, t_2) \leq t_2 - t_1$.

$dbf\,(t_1, t_2)$ denotes the *processor demand bound function* and is given by:

$$dbf\,(t_1, t_2) = \sum_{i=1}^{n} \eta_i\,(t_1, t_2)\, C_i. \tag{1}$$

where $\eta_i\,(t_1, t_2) = \max\left(0, \lfloor \frac{t_2 - \phi_i - D_i}{T_i} \rfloor - \lceil \frac{t_1 - \phi_i}{T_i} \rceil + 1\right)$ represents the number of jobs of task τ_i which occur in the interval $[t_1, t_2)$ with a deadline less than or equal to t_2.

In the KTS partitioned mapping approach (presented hereafter), Lemma 1 is applied to a given core of the NoC locally, in order to test if a new task can be mapped or not to the core, checking the feasibility of the set of real-time tasks formed by the ones already mapped to that core plus the new task.

4 K-level task splitting (KTS)

We present a new static mapping algorithm called K-level task splitting (KTS) based on partitioned scheduling with a task splitting phase. Some tasks are mapped to single processors, while other ones are divided into asynchronous subtasks with smaller utilization factor, each being mapped to an individual processor. KTS consists of a single *mapping phase* in which each task is mapped to a specific processor in the NoC.

This approach is deemed a partitioning scheme rather than a semi-partitioning scheme. It permits the different jobs generated by a given task to be executed on different cores but each job is executed on one core only. Hence, it makes sense to treat the proposed approach as a *static mapping* one for the two following reasons:

1. Every subtask is still a periodic task whose jobs are activated less frequently but, unlike other methods, jobs themselves are not divided among different processors (i.e. only successive task invocations are spread over different processors);
2. Task code being identical for each subtask, it can be replicated, and since jobs of a given task are assumed to be independent of each other, data migration for subtasks is not required at job boundaries.

The only requirement of the algorithm is that all processor clocks in the NoC be synchronized in order to ensure the subtask scheduling order. As reviewed in Sect. 2.2, deterministic clock synchronization solutions for many-core topologies exist and are suitable for hard real-time systems [9]. KTS is assumed to be disseminated in the form of an open-source real-time partitioning tool in order to be used in total independence with the real-time operating system running on the cores of the NoC.

4.1 Principle

KTS first attempts to map each task to a particular processor through a bin-packing heuristic (e.g. FFD, BFD…). If a task is rejected by all processors, it will be split into two subtasks. The resulting tasks will have smaller utilization factor (i.e. half the value of the task's utilization factor) and different offsets so as to be mapped to different processors and executed without any overlapping in time.

The algorithm then attempts to map those resulting tasks to a processor. The exact schedulability test for periodic tasks with offsets described in Lemma 1 is used. If a subtask cannot be mapped to any single processor, it is split again. Each splitting step corresponds to a level. We define a depth limit K, that is to say, a task cannot be split more than K times. K is given by the user. Even if K is too large, KTS may stop splitting a task before reaching the depth limit K if all tasks are successfully mapped to cores. This means a task does not necessary need to be split K times. If K is very large, the success of mapping the whole task set is maximized but at the cost of an increased computation time (the algorithm is recursively called each time a task is split).

More formally, a task $\tau_i(\phi_i, C_i, T_i, D_i)$ being rejected by all processors is split into two subtasks $\tau_i'(\phi_i, C_i, 2*T_i, D_i)$ and $\tau_i''(\phi_i + T_i, C_i, 2*T_i, D_i)$, as depicted in Fig. 2. Provided the processor clocks of the NoC be synchronized, subtasks will be resumed at the "right" time (task t_i'' with a given offset), thus guaranteeing that the activation of the subtasks τ_i' and τ_i'' be equivalent to the activation pattern of the original task τ_i.

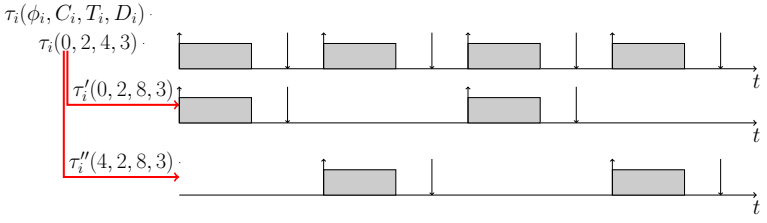

Fig. 2 Task splitting

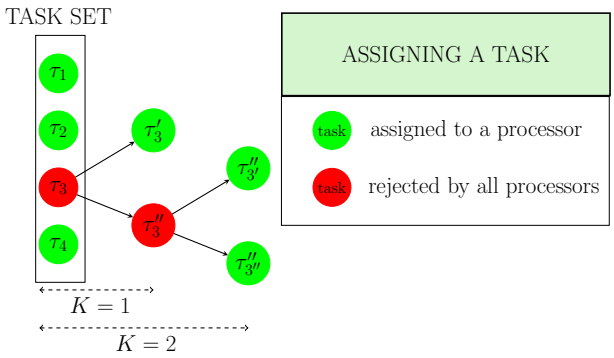

Fig. 3 KTS's consecutive task splittings according to parameter K

Figure 3 illustrates the algorithm for $K = 2$. KTS allows up to two levels of derivations for each task. At level 0, task τ_3 cannot be mapped to a single processor. So, it is divided into two tasks τ'_3 and τ''_3 at the first level ($K = 1$). Then one of the two subtasks (τ''_3) is rejected. τ''_3 is then split again to derive two other tasks at level 2. The algorithm continues until either it reaches the depth limit K or all tasks are successfully mapped. In the example depicted in Fig. 3, the algorithm reaches the limit depth $K = 2$ in which all the subtasks are successfully mapped. Consequently, the task set is schedulable on the given NoC.

More generally, on a NoC-based platform composed of m cores, every task can be potentially split K times into two subtasks of lower processor utilization factors. However, if all cores are already loaded with tasks in such a way that their remaining capacity is less than the processor utilization factor of the original task to be mapped divided by m, then the task will inevitably be rejected (condition 1. of Lemma 1 will be violated).

Let us note that KTS algorithm with $K = 0$ is equivalent to the fully partitioning.

Figure 4 shows the pseudo-code of KTS with FFDD (First Fit Decreasing Density) bin-packing heuristic. $current\_k$ represents the current level of derivation of subtasks, while K is the depth limit for task splitting. In case of successful mapping to processors, the algorithm returns SUCCESS with the corresponding mapping of task set $\mathcal{T}$ on processor set Π, denoted $map(\mathcal{T}, \Pi)$. Otherwise, it returns FAILURE. In this case, task set $\mathcal{T}$ is said to be unfeasible since some tasks have been rejected. As the objective is to execute the whole task set on the NoC, even if the algorithm leads to a partial mapped task set, if it returns FAILURE, no task is executed.

For any processor π_j, let $map(\pi_j)$ denote the tasks among $\tau_1,\ldots,\tau_{i-1}$ that have already been mapped to processor π_j. Initially, $map(\pi_j) = \emptyset$. At the beginning, the algorithm performs using an FFDD algorithm. If no such π_j exists, then we are unable to conclude that the periodic task set $\mathcal{T}$ is feasible upon the m-processor NoC. Task τ_i is then split into a set $\tau_i^{split} = \{\tau'_i(\phi_i, C_i, 2\times T_i, D_i); \tau'_i(\phi_i + T_i, C_i, 2\times T_i, D_i)\}$ of two subtasks that are mapped to a subset of processors belonging to the NoC (at most m).

KTS is a recursive algorithm. First call of KTS is with the value $current\_k = 0$. If at some point a task is rejected by all cores, it is split and the split tasks belong to the next level $current\_k+1$. KTS then tries to map them by calling itself for the next level

Algorithm 1 KTS Partitioned Algorithm

Input: $m, current\_k, K, \mathcal{T} = \{\tau_1, ..., \tau_n\}$

Output: SUCCESS and $map(\mathcal{T}, \Pi)$ if $\mathcal{T}$ is feasible, FAILURE otherwise.

Function: KTS(in m, in $current\_k$, in K, in $\mathcal{T}$, out $map(\mathcal{T}, \Pi)$)

```
begin
for i = 1 → |τ| do
    Sched ← false;
    /* mapping the task according to FFDD */
    for j = 1 → m do
        if τ_i feasible on π_j then
            /* τ_i is mapped to π_j */
            map(π_j) = map(π_j) ∪ τ_i;
            Sched ← true;
            break;
        end if
    end for
    if not Sched then
        /* Trying to split the task */
        if current_k < K then
            /* the depth limit K has not been reached */
            split τ_i(φ_i, C_i, T_i, D_i) into a new set of two subtasks;
            τ_i^split = {τ'_i(φ_i, C_i, 2 × T_i, D_i); τ'_i(φ_i + T_i, C_i, 2 × T_i, D_i)};
            if KTS(m, current_k + 1, K, τ_i^split) == SUCCESS then
                Sched ← true;
            end if
        else
            /* the depth limit K has been reached*/
            Sched ← false;
        end if
    end if
    if not Sched then
        return FAILURE;
    end if
end for
map(T, Π) = ⋃_{j=1}^{m} map(π_j)
return SUCCESS;
end
```

Fig. 4 Pseudo-code of KTS

$current\_k = current\_k + 1$ and so on until all tasks are successfully mapped to cores or until the current level $current\_k$ reaches the max depth K without mapping all the tasks. If all tasks and resulting split tasks are successfully mapped to cores, the task set is feasible. Otherwise, if K is reached without mapping all tasks, it is not feasible.

4.2 Properties

The following lemma asserts that, in mapping a subtask of τ_i^{split} to a processor π_j, the partitioning algorithm does not adversely affect the feasibility of the tasks mapped earlier to processors.

Lemma 2 *If tasks previously mapped to processors were feasible (on each processor) and the algorithm above maps τ_i' (respectively, τ_i'') to processor π_j (according to Lemma 1), then tasks mapped to any processor (including processor π_j) remain feasible.*

Proof Sketch Observe that the feasibility of processors other than processor π_j is not affected by the mapping of task τ_i' (respectively, τ_i'') to processor π_j. If the tasks mapped to π_j were feasible on π_j prior to the mapping of τ_i' (respectively, τ_i'') and condition of Lemma 1 is satisfied, then the tasks on $map(\pi_j)$ remain feasible after adding τ_i' (respectively, τ_i''). □

The correctness of the partitioning algorithm follows, by repeated applications of Lemma 2.

Theorem 1 *KTS partitioning algorithm returns SUCCESS on task set $\mathcal{T}$ if and only if the resulting partitioning is feasible.*

Proof Sketch Observe that the algorithm returns SUCCESS if and only if it has successfully mapped each task in $\mathcal{T}$ to some processor. Prior to the mapping of task τ_i, each processor is trivially feasible. It follows from Lemma 2 that all processors remain feasible after each task mapping as well. Hence, all processors are feasible once all tasks in $\mathcal{T}$ have been mapped. □

Theorem 2 *KTS performance stops to increase when the first job of a task cannot be executed on any processor without causing deadlines violation.*

Proof Sketch A task cannot be mapped to any processor even if its utilization factor is less than the spare capacity due to the fact that its first job cannot be executed successfully on any processor. Considering the task splitting scheme Fig. 2, splitting the task will not split the job execution time and subsequently resulting tasks will remain unmapped to any processor. For that reason, adding a level of splitting is useless. □

From the pseudo-code given in Fig. 4, we also derive that the complexity of KTS is quadratic polynomial (see Theorem 3).

Theorem 3 *KTS has order of n^2 time complexity.*

Proof Sketch The mapping problem for NoC is to decide how to place a set of n hard real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ onto the cores of a given network such that all timing constraints are met :

- If $n \leq m$, no need for task splitting and in the worst case each task if assign to a different core.
  - with KTS-FF : the first task passes the schedulability test on first core and is assigned to it (1 operation), the second task in the worst case is assigned to the second core after being rejected from first core which means the schedulability test is done two times (two operations), …, the n-th task in the worst case is assigned to the n-th core n after being rejected from previous core which means the schedulability test is done n times (n operations), which leads us to say that the total number of operations is $1 + \cdots + n$ that to say $n(n+1)/2$.

  - with KTS-WF : each task does the schedulability test on all of the m cores before choosing the one that can receive the task while maximizing its spare capacity. The total number of operations is $n \times m$.
- If $n > m$:
  - with KTS-FF : the m first tasks pass their schedulability test as described in the case $n \leq m$ ($1 + 2 + \cdots + m = (m+1)m/2$ operations) and are assigned with success to cores. Each of the rest of tasks $(n - m)$, in the worst case, tries to be assigned to the m cores ($(n - m)m$ operations). If it is rejected by all cores, it is split into two subtasks at each splitting operation. At each level k they double their number before re-testing on the m cores ($(n - m)m.2 + (n - m)m.2^2 + \cdots + (n - m)m.2^K$ operations). We can deduce that the total number of operations is $m(m+1)/2 + (n - m)m \times K_s$ avec $K_s = \sum_{k=0}^{K} 2^k$ which is a constant value.
  - with KTS-WF : the m first tasks pass their schedulability test as described in the case $n \leq m$ ($1 + 2 + \cdots + m = (m+1)m/2$ operations) and are assigned with success to core(m^2).Each of the rest of the tasks tries to be assigned to the m cores ($(n - m)m$ operations). If they are rejected by all cores, they are split (two resulting subtasks at each splitting operation) at each level k before they all are assigned. At each level k they double their number before re-testing on the m cores ($(n - m)m.2 + (n - m)m.2^2 + \cdots + (n - m)m.2^K$ operations). The total number of operations is $m^2 + (n - m)m \times K_s$ avec $K_s = \sum_{k=0}^{K} 2^k$ which is also a constant value.

As $n = \beta m$ with $\beta \epsilon \mathbb{R}^+$, KTS complexity is in $O(n^2)$. □

5 Performance analysis and discussion

In this section, we analyse the effectiveness of KTS with respect to classical FFDD bin-packing heuristic and EDF Split (DD) algorithm [6].

5.1 Simulation set-up

In our task generation methodology, we consider two sets of experiments. The first one is relative to tasks with implicit deadlines. The second one is relative to tasks with constrained deadlines as described in what follows:

- Task utilization factors u_i are randomly generated (uniform distribution) in the range [0.1, 1].
- Task periods T_i are picked in the interval [20, 200];
- Task WCETs C_i are computed from periods T_i and utilization factors u_i: $C_i = T_i.u_i$
- Task relative deadlines D_i are randomly chosen in the range $[C_i, T_i]$;

We generate 100 different task sets on a m-processor NoC with $m \in \{16, 32, 64, 128\}$.

A task set is said to be successfully scheduled if all tasks in the set are successfully mapped to processors. The effectiveness of the algorithm KTS is measured by the *Success_Ratio*:

$$Success\_ratio = \frac{N_{success}}{N_{generated}} \tag{2}$$

where $N_{success}$ and $N_{generated}$ are the number of successfully scheduled task sets and the total number of task sets, respectively.

In all experiments, we generate 100 task sets with normalized system utilization $u_{sys} \in [0.6, 1]$. Task sets are composed of $2 \times m$ tasks with either implicit ($D = T$) or constrained deadlines ($D \leq T$).

5.2 Simulation results

KTS-FFDD ($K = x$) refers to KTS with a depth limit $K = x$ based upon the KTS-FFDD heuristic. It is compared to both EDF Split (DD) and the classical heuristic FFDD.

5.2.1 Experiment 1: Effect of the number of processors in the NoC

Figure 5 illustrates the comparative performance of FFDD, KTS-FFDD and EDF Split (DD) with varying the number of processors m of the NoC for *implicit-deadline* task sets.

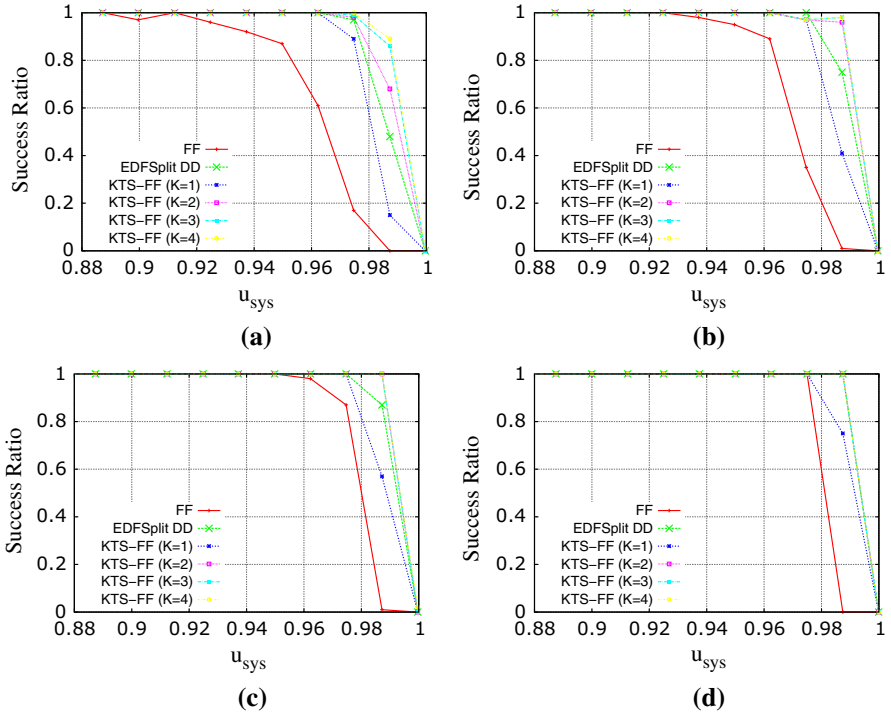

Fig. 5 Success ratio for $D = T$ with: **a** $m = 16$, **b** $m = 32$, **c** $m = 64$, **d** $m = 128$

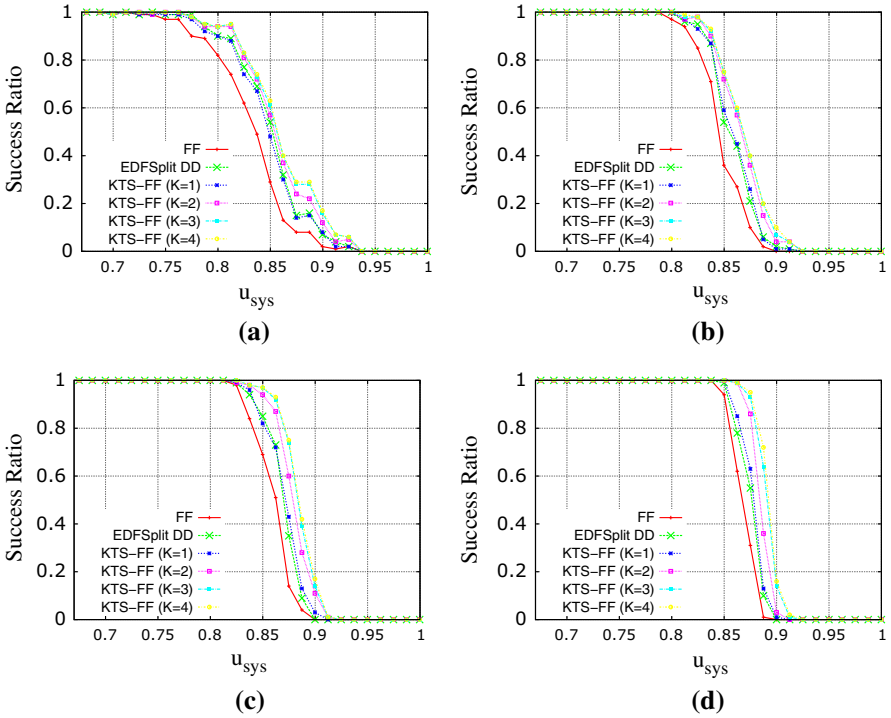

Fig. 6 Success ratio for $D \leq T$ with: **a** $m = 16$, **b** $m = 32$, **c** $m = 64$, **d** $m = 128$

We note that, as the number of processors increases, the performance of all strategies increases. The success ratio observed for both KTS-FFDD and EDF Split (DD) clearly exceeds that of FFDD in all observed cases. As expected, the higher the value of K, the higher the success ratio. KTS-FFDD always behaves better than EDF Split (DD) as soon as $K > 1$. For instance, for $u_{sys} = 0.986$ and $m = 32$, EDF Split (DD) only schedules 75 % of the task sets generated while 98 % are schedulable using KTS-FFDD with $K = 4$. It is worth noticing that for the case $m = 128$ (i.e. for a large number of processors in the NoC), KTS-FFDD and EDF Split (DD) achieve similar performance. However, KTS-FFDD does not imply any migration cost at run-time, thus still being the best solution to choose.

Figure 6 depicts the curve of the success ratio for the same cases (i.e. NoCs composed of 16, 32, 64 and 128 processors) in considering *constrained-deadline* task sets.

Here, the overall success ratio for all algorithms is not as high as in the case $D = T$. Our algorithm always outperforms the other algorithms if $K > 1$. In contrast to the case with implicit-deadline tasks, EDF Split (DD) never reaches the good performance of KTS-FFDD even for a high number of processors. For $u_{sys} = 0.875$ and $m = 128$, FFDD schedules 31 % of the task sets, while 35 % of them are schedulable under EDF Split (DD) and 95 % under KTS-FFDD with $K = 4$.

The very good performance of KTS-FFDD can be explained easily: the higher the number of processors, the higher the chance of splitting tasks into a large number of subtasks with low utilization factors, thus mapping them more easily into the NoC.

5.2.2 Experiment 2: Effect of light/heavy tasks mapped into the NoC

In this experiment, we study the effect of task utilization factors on the effectiveness of KTS-FFDD, EDF Split (DD) and FFDD on a 64-processor NoC. A task with utilization factor less than 0.5 is called a *light* task, while a task with utilization factor at least 0.5 is called a *heavy* task.

We consider three profiles of task sets. If all utilization factors are drawn from [0.1; 0.5], [0.1; 1] and [0.5; 1], we, respectively, refer to *light distribution*, *medium distribution* and *heavy distribution*.

Success ratios for *light, medium and heavy distributions* are depicted with respect to normalized system utilization in Figs. 7, 8, and 9, respectively. Both implicit- and constrained-deadline cases are considered.

Results clearly indicate that both KTS-FFDD ($K = 4$) and EDF Split (DD) are the best heuristics whatever is the case under consideration. Moreover, we note that the higher the ratio of heavy tasks in the task set, the more efficient are EDF Split and KTS-FFDD in comparison to FFDD. KTS-FFDD with $K > 1$ outperforms EDF Split (DD) for medium distributions only, but the gain is quite significant: for constrained-deadline tasks, when utilization factors are in the range [0.1, 1] and $u_{sys} = 0.875$, only

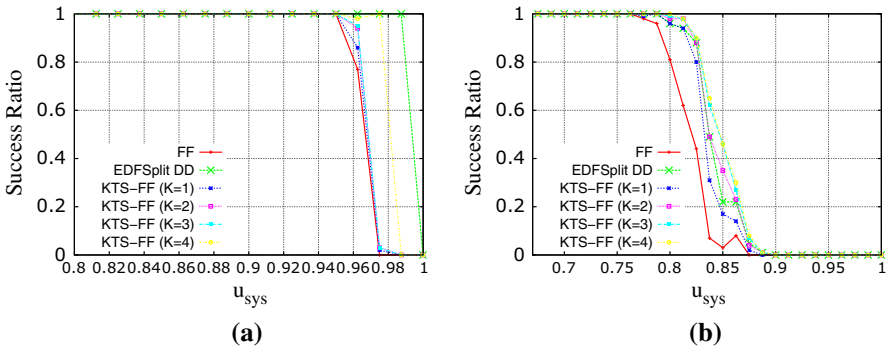

Fig. 7 Success ratio for *light distributions*: **a** $D = T$, **b** $D \leq T$

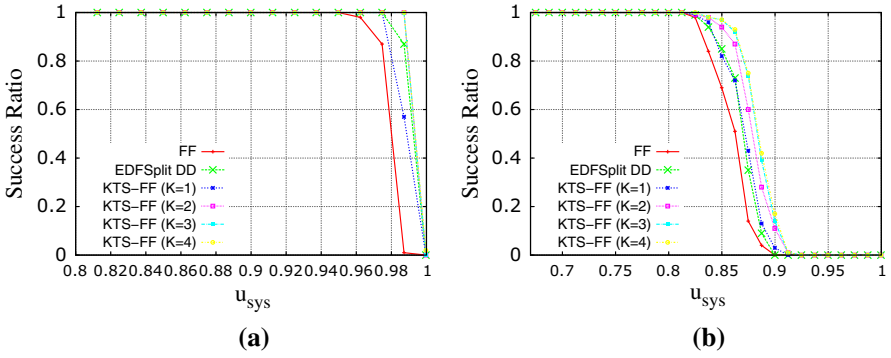

Fig. 8 Success ratio for *medium distributions*: **a** $D = T$, **b** $D \leq T$

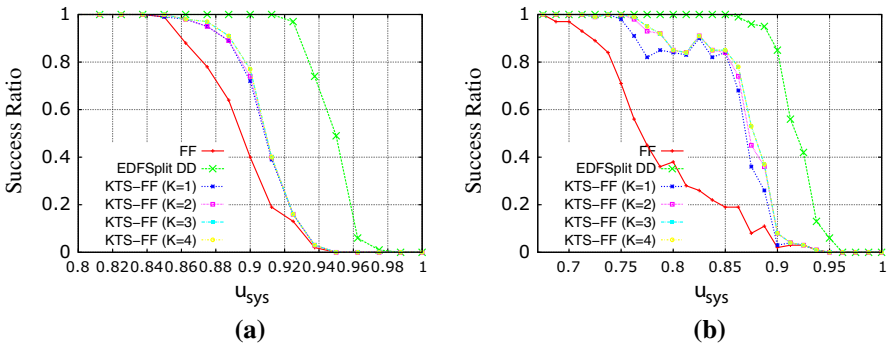

Fig. 9 Success ratio for *heavy distributions*: **a** $D = T$, **b** $D \leq T$

Table 2 Cross-comparison of the mapping algorithms

Algorithms	Performance	Computational complexity (offline)	Temporal overhead (online)
FFDD	Poor	Linear	None
EDF Split (DD)	Good	Pseudo-polynomial	Quite significant
KTS-FFDD ($K > 1$)	Very good/tunable	Quadratic polynomial	None

35 % of the task sets are schedulable using EDF Split (DD) while 75 % are schedulable using KTS-FFDD with $K = 4$.

5.3 Synthesis and discussion

Evaluation over a wide range of randomly generated task sets indicates that KTS-FFDD algorithm is a good alternative to EDF Split (DD) algorithm. KTS approach is conceptually simple, easy to implement in existing systems and effective. Every subtask is still a periodic task whose code can be replicated.

Table 2 gives a qualitative evaluation of the three mapping algorithms under consideration in terms of performance, computational complexity and temporal overhead.

It supports the conclusion that EDF Split (DD) and KTS-FFDD offer benefits over conventional partitioned FFDD approach in terms of performance (i.e. the ratio of the number of successfully mapped task sets over the total number of submitted task sets is higher), but not both algorithms give good results at run-time. Aside from its requirement of processor clocks in the NoC to be synchronized, KTS-FFDD does not require any temporal overhead (i.e. no additional algorithm is required at run-time in order to ensure the task jobs precedence constraints). This is not the case for EDF Split (DD). Important for the selection of a mapping strategy is not only performance, but also ease of implementation and application. EDF split (DD) will require more implementation effort to be deployed. The downside of the zero run-time overhead of KTS-FFDD is that the algorithm exhibits a higher (but also polynomial) complexity for the offline mapping phase. According to the level of performance required by the

application, it is always possible to tune the split depth limit K value so as to reach a good compromise between performance and computation time.

Finally, compared to EDF Split (DD), KTS cannot handle sporadic tasks (i.e. tasks for which period is interpreted as a *minimum* inter-arrival interval). However, KTS can be implemented in many applications, whereas EDF Split (DD) can only be used in systems in which the code of each task may be explicitly divided into two parts mapped on two different processors. This is not always the case, and when appropriate, it requires an in-depth WCET analysis of the code.

6 Conclusion

This paper addressed an important issue for the practical utilization of many-core architectures based on NoCs for real-time systems: mapping of deadline-constrained tasks. We introduced a novel static mapping algorithm based on task splitting. We showed how the classical real-time feasibility analysis for asynchronous tasks can be used in such a task splitting scheme. The competitive schedulability of KTS is achieved without any additional number of context switches and without any migration cost, thus resolving the problem of the global intrachip communication costs that can highly degrade the overall performance of the NoC. The only requirement of the approach is that all processor clocks be synchronized.

Future works include extending this approach to firm real-time systems that allow some of their jobs to be occasionally discarded so as to handle transient overloads.

References

1. Anderson J-H, Bud V, Devi U-C (2005) An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In: Proceedings of the 17th IEEE Euromicro Conference on Real-Time Systems, pp 199–208
2. Anderson J-H, Erickson J-P, Devi U-C, Casses B-N (2014) Optimal semi-partitioned scheduling in soft real-time systems. In: Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications
3. Baruah S-K, Rosier L-E, Howell R-R (1990) Algorithms and complexity: concerning the preemptive scheduling of periodic real-time tasks on one processor. Real-Time Syst J 2(4):301–324
4. Bjerregaard T, Sparsø J (2005) Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems, pp 34–43
5. Buckler M (2014) Network-on-chip synchronization. Ph.D. thesis, University of Massachussetts
6. Burns A, Davis R-I, Wang P, Zhang F (2012) Partitioned EDF scheduling for multiprocessors using a C=D scheme. Real-Time Syst J 48(1):3–33
7. Chapiro D-M (1984) Globally asynchronous locally synchronous systems. Ph.D. thesis, Stanford University
8. Hilbrich R, Van Kampenhout JR (2011) Partitioning and task transfer on NoC-based many-core processors in the avionics domain. Softwaretech Trends J Softwaretechnik-Trends 30(3):6
9. Kasapaki E, Schoeberl M, Sorensen R-B, Muller C, Goossens K, Sparso J (2015) Argo: a real-time network-on-chip architecture with an efficient GALS implementation. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(2):479–492
10. Leung J-Y-T, Merill M-L (1980) A note on preemptive scheduling of periodic. Real-Time Tasks Inf Process Lett 11(3):115–118
11. Leung J-Y-T, Whitehead J (1982) On the complexity of fixed-priority scheduling of periodic real-time tasks. Perform Eval 2:237–250

12. Lupu I, Courbin P, George L, Goossens J (2010) Multi-criteria evaluation of partitioning schemes for real-time systems. In: Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation
13. Martin A, Nystrom M (2006) Asynchronous techniques for system-on-chip design. Proc IEEE 94(6):1089–1120
14. Milojicic D, Douglis F, Paindaveine Y, Wheeler R, Zhou S (2000) Process migration survey. ACM Comput Surv 32(8):241–299
15. Palesi M, Daneshtalab M (2014) Routing algorithms in networks-on-chip. Springer, New York
16. Sparso J (2004) Future networks-on-chip; will they be synchronous or asynchronous? In: Proceedings of the Swedish System on Chip Conference. Bastad, Sweden, pp 13–14
17. Stankovic J-A (1988) Misconceptions about real-time computing: a serious problem for next-generation systems. Computer 21(10):10–19
18. Tatas K, Siozios K, Soudris D, Jantsch A (2014) Designing 2D and 3D network-on-chip architectures. Springer, Berlin