Network-on-Chip Aware Scheduling of Hard-Real-Time Tasks

Mayank Shekhar[1], Harini Ramaprasad[1], Frank Mueller[2]
mayank@siu.edu, harinir@siu.edu, mueller@cs.ncsu.edu
[1]Southern Illinois University Carbondale, [2] North Carolina State University

Abstract— **As real-time systems continue to integrate more and more functionality, powerful multi-core architectures are the only viable solution to meet their computational demands under reasonable energy budgets. Dramatic increases in the number of cores on multi-core architectures have led to scalability issues. Modern platforms are moving away from designs with shared caches and shared buses to designs with private caches on cores and high-bandwidth Networks-on-Chip (NoC) for communication. On such architectures, worst-case execution times of real-time tasks depend on the physical location of cores and interference on the NoC. In this paper, we present a dynamic-priority policy for scheduling memory accesses on the NoC and a location-aware partitioning policy that takes explicit advantage of this NoC scheduling policy to improve the efficiency of task allocation. We demonstrate that this combination achieves significant improvement in task set schedulability and NoC utilization.**

I. Introduction and Related Work

The need to meet ever-increasing computational demands while still maintaining reasonable power/energy consumption led to the advent of multi-core architectures. Multi-core architectures were initially used predominantly for general-purpose computing. However, as real-time systems integrate more and more functionality, they are demanding more computational power, making multi-core architectures a natural choice even within the real-time systems domain. Real-time systems are those where predictability and timing of functionality (i.e., tasks) are paramount. While ensuring predictability and temporal correctness on single-core architectures is challenging enough, the use of multi-core architectures for real-time execution introduces additional challenges.

A fundamental consideration in ensuring safe and efficient execution of real-time tasks on multi-core architectures is how to schedule tasks on cores. Real-time scheduling on multi-core architectures has been the focus of much research and most techniques fall into three categories, namely partitioned scheduling, global scheduling and a hybrid of the two, semi-partitioned scheduling. In partitioned scheduling, tasks are statically allocated to cores and are scheduled using single-core scheduling algorithms on each core. Partitioned scheduling is equivalent to a bin-packing problem and may be solved in practice with variants such as first fit, best fit, worst fit, etc. In global scheduling, all tasks are placed in a common queue and dynamically scheduled on available cores, potentially resulting in task migrations. The third category is a hybrid of the first two and is called semi-partitioned scheduling. In this case, most tasks are partitioned onto cores and only a few of them are allowed to migrate, typically in a pre-determined fashion. Davis et al. recently conducted a comprehensive survey of hard-real-time scheduling techniques for multi-core architectures [1].

This work was supported in part by NSF grants CNS-0905212, CNS-0720496 and CNS-0905181.

A. Architectural Considerations

Foremost among factors that impede predictability on multi-core architectures are on-chip memory hierarchies and the on-chip communication infrastructure. Most of the scheduling techniques discussed in the above-mentioned survey [1] do not explicitly consider the effects of such architectural factors, including cache conflicts among tasks scheduled on a given core, overheads incurred due to cache content migration (in the case of non-partitioned scheduling approaches), contention on the shared communication infrastructure among migration traffic and main memory traffic, etc. They assume that the effects of these factors are either negligible or that they are orthogonal aspects that can be easily subsumed into worst-case execution time (WCET) analysis of a task.

Consideration of architectural factors has been addressed in several ways. A recent approach by Yao et al. [2] proposes a memory centric scheduling approach for hard-real-time tasks executing on a shared bus based multi-core architecture. The fundamental idea is to divide tasks on each core into memory phases (phase where memory requests are issued) and execution phases (phase where execution is performed using cached data) and employ Time Division Multiple Access (TDMA) for memory accesses issued by multiple cores. The main drawback of this approach is the fact that it requires a transformation of tasks that may not be practical for all types of tasks.

Consideration of shared caches and shared buses as part of WCET analysis has been explored by several researchers. Yan and Zhang [3] propose techniques to calculate the WCET of tasks on multi-core architectures with shared caches. Chattopadhyay et al. [4] propose a timing analysis technique for multi-core architectures that considers both shared caches and shared buses. Kelter et al. [5] discuss shared bus-aware WCET analysis under a TDMA approach.

In recent architectures, on-chip memory hierarchies are moving away from designs with shared caches for (groups of) cores and towards designs with only private caches on each core. As the number of cores on a single chip continues to increase, in the interest of scalability, shared bus-based architectures are giving way to platforms with a Network-on-Chip (NoC) for communication among cores and with off-chip main memory. Examples of such architectures include the Intel SCC platform [6] and Tilera's scalable architectures such as the TilePro64 platform [7]. Recently, several researchers have proposed techniques targeted at such scalable architectures.

The area of scheduling traffic on the Network-on-Chip (NoC) has been the focus of much research in recent years, with techniques being proposed for both general purpose and real-time computing. Bjerregaard et al. conduct a survey on research practices of NoCs in general [8]. We now specifically discuss work relating to handling NoC traffic (e.g., NoC scheduling, routing, etc.) There are primarily two types of approaches that have been proposed to handle NoC traffic, namely static resource reservation-based approaches and approaches using run-time arbitration of resources.

Goossens et al. [9] and Millberg et al. [10] propose approaches where the available link transmission capacity is partitioned into fixed time-slots and a particular traffic flow occupies the physical link in each time slot. Circuit-switching techniques were proposed by Wolkotte et al. [11] and Wiklund and Liu [12]. Here, a dedicated path is provided for a source and destination core pair that cannot be used by any other cores. In previous work, we propose a simple time division multiplexing based approach, namely weighted TDMA [13]. In this work, our goal is to handle contention among main memory traffic and to make task WCET independent of the physical location of the core to which it is allocated. The advantages of resource reservation-based approaches is their simplicity and ease of analysis. On the other hand, the fundamental drawback is under-utilization of the NoC.

Bolotin et al. [14] propose QNoC, a technique that divides the network services into four different levels and performs a prioritized arbitration of resources according to the level of service available in the system. Kavaldjiev et al. [15] present a round-robin approach to schedule real-time services on the system. Bjerregaard and Sparso [16] propose a priority-based round-robin technique to bound latency and bandwidth. Priority-based wormhole switching technique was introduced by Shi and Burns [17] to provide communication service guarantees. In this work, traffic flows are assigned priorities and contention is resolved on the basis those priorities. The limitation of this approach is that as the number of traffic flows increases, it becomes impractical to maintain separate priorities for all of them. To overcome this limitation, Shi et al. [18] proposed a priority share algorithm where multiple traffic flows share the same priority. Shi et al. [19] also proposed a new 'per-priority' analysis approach that can efficiently handle wormhole switching with a priority sharing policy. In this work, they also present a task mapping and priority assignment-based on the priority sharing policy in order to meet the real-time requirements.

Nikolic et al. [20] propose worst case off-chip memory traffic analysis under a limited migrative model. They allocate tasks onto cores and propose a technique to calculate the worst-case interference time of packets belonging to a given task. This work assumes static priority assignment for tasks.

All the techniques for scalable multi-core architectures discussed thus far assume that task allocation onto cores is known *a-priori*. In contrast, we argue that cache and NoC aspects must be considered *during* the task allocation phase in an effort to derive task allocations that improve cache and NoC characteristics.

To this end, in recent work, we propose two schemes. First, we propose a partitioned real-time scheduling scheme [21] in which each task is allowed to lock a subset of its memory lines in the cache on the core to which it is allocated[1]. In order to maximize cache locking, task allocation is performed with the explicit goal of reducing cache conflicts among tasks on a given core. Second, we propose an architecture-aware semi-partitioned scheduling scheme [13] that also allows tasks to lock a subset of memory lines in cache. This scheme employs a proactive cache content migration scheme [27] to migrate locked cache lines when a task migrates, explicitly considering the overheads of this migration.

Both the above schemes assume that memory access latencies for a task are independent of the physical location of the core on which it is allocated. For example, in order to enforce this, we propose and employ a weighted TDMA-based NoC scheduling policy in our architecture-aware semi-partitioned scheduling scheme [13]. However, in the process of making memory access latency location-independent, we increase pessimism and sacrifice NoC performance.

B. Contributions of this Paper

In this paper, we propose a two-part scheduling scheme for hard-real-time tasks executing on cache-based architectures with NoCs. The first part is a policy for memory traffic scheduling on the NoC where the memory access latency for a core is proportional to its distance from the memory controller. The second part is a location-aware partitioning task allocation policy that allocates tasks with the explicit goal of maximizing cache locking benefits under the given NoC scheduling policy. We compare the performance of our algorithm with that of our previous architecture-aware semi-partitioned scheme [13] and demonstrate that it is better, both in total schedulable utilization of the cores and in utilization of the NoC[2].

II. Assumptions and System Model

In this section, we present assumptions made in our work and introduce the terminology used in the remainder of this paper.

We assume a homogeneous multi-core architecture where each core is assumed to have a lockable, set-associative private cache and it is assumed that there are no shared caches. We assume a hard-real-time sporadic task model where relative deadlines of tasks are less than or equal to their minimum inter-arrival times (hereafter called period). A task τ_i is represented by the tuple (T_i, C_i, D_i), where T_i is its period, C_i is its base worst-case execution time (WCET) and D_i is its relative deadline. We assume that a task may lock its memory footprint in the cache on its core and that every task requires at most one way in each cache set. The base WCET of a task, namely C_i, is calculated assuming that a task is able to lock its *entire* memory footprint in the cache on its core. The base utilization of a task τ_i, denoted by $Util_i$, is defined as the ratio of its base WCET to its period. Tasks are assumed to be independent in

[1]Cache locking is a commonly used technique to improve the predictability of cache behavior. Several techniques [22], [23], [24], [25], [26] have been proposed in recent years for static and dynamic cache locking.

[2]In order to perform a fair comparison of our proposed approach to other existing NoC scheduling and analysis schemes such as the Limited Migrative Model approach proposed by Nikolic et al. [20], significant adaptation of existing schemes is required due to different assumptions about task allocation and cache-related aspects. Such adaptation is out of the scope of the current paper. However, we will explore this as part of future work.

nature. Tasks on a given core are assumed to be scheduled using an Earliest Deadline First (EDF) policy.

We assume that the architecture includes a mesh-based Network-on-Chip (NoC) with worm-hole switching for communication. We assume that separate channels are available in the NoC for communication among cores and communication between cores and main memory, as is found in Tilera's TilePro64 architecture [7]. We assume that there are multiple memory controllers, each with multiple ports, arranged around the mesh of cores. For example, the TilePro64 architecture has 64 cores arranged in a 2-D mesh and four memory controllers, arranged two above and two below the mesh. We assume that every memory controller can access the entire main memory, as is the case on TilePro64 when the memory is not striped. Each memory controller has four ports. Figure 1 shows the layout of one quadrant of the architecture assumed in this paper. Each port is assumed to serve four cores in a single column (e.g., Port1 serves cores A, B, C and D). So, every off-chip memory request is routed along a straight path to the appropriate memory controller and a specific port within the memory controller, as depicted in Figure 1.

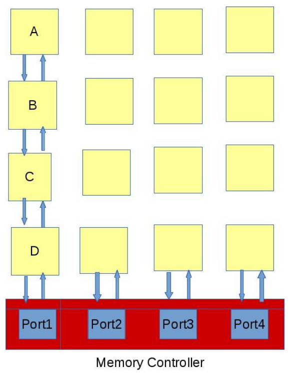


Fig. 1. 4-core shared bus Tilera like architecture

III. Methodology

The contributions of this paper are two fold. The first is a policy for scheduling memory traffic on the NoC and the second is a location-aware partitioning of tasks onto cores. Our fundamental idea for memory traffic scheduling on the NoC is to control the rates of memory requests issued by different cores to different pre-calculated values and schedule the memory transfers over a predetermined set of NoC links using a dynamic priority scheduling policy. Under this scheme, memory requests from different cores experience different memory transfer latencies depending on the core's distance from the memory controller. The goal of the location-aware partitioning phase is to derive schedulable task allocations, if possible, and assign memory request rates to each core in the process. Note that our scheme does not require any modification to user-level tasks executing on cores. Once a memory request has been issued, a task simply busy waits until the request is completed, as is typical. Rate controlling is performed transparently at the system level.

As mentioned in Section II, we assume that all memory traffic from a given core travels along a fixed, straight path to a designated port of a designated memory controller. For example, in Figure 1, all cores (in the quadrant shown) are served by the single memory controller shown and core D can take only the highlighted route to access (off-chip) memory and can only be served by Port1 on the memory controller. By employing this approach, we essentially bound the number of cores sharing a given column of NoC links and a given port within a given memory controller. In the remainder of this section, we explain the details of our algorithm using a single column of (four) cores that share one port in the memory controller since traffic flows from multiple such columns do not interfere with each other in our scheme.

A. *Memory Traffic Scheduling Terminology*

Memory traffic is scheduled on the NoC in a manner analogous to real-time task scheduling on a processor and we employ a policy analogous to the Earliest Deadline First (EDF) scheduling policy for this purpose since EDF supports maximal utilization. We name our NoC scheduling policy **EDF-on-NoC**. Memory requests from a core are analogous to jobs of a sporadic real-time task and the NoC bandwidth is analogous to the processor on which these jobs are to be scheduled. In accordance with this analogy, every core k is assumed to contain one abstract *implicit deadline* sporadic task, namely τ_M^k, with minimum inter-arrival time or period represented by T_M^k and a worst-case execution time or WCET represented by C_M^k. T_M^k represents the minimum separation between consecutive memory requests that our algorithm *enforces* on a given core k. C_M^k represents the worst-case *on-chip* latency of a memory access initiated by core k when it is the *only* memory access in progress over a given NoC column. C_M^k for a given core k is calculated based on the distance of the core from the memory controller designated to serve the core. T_M^k is calculated such that it is larger than or equal to the worst-case response time (i.e., the time between the initiation and completion) of a memory access initiated by core k, but small enough to ensure schedulability of tasks on core k. We now describe the calculation of C_M^k. Since the calculation of T_M^k is dependent on task allocation, it will be described later.

B. *Calculation of C_M*

Memory requests may be of two types, namely *read* and *write* requests. In the case of a read request, the memory request for a particular cache line goes from the core to the memory controller and then a memory line is transferred from the memory controller to the core over a NoC column. For a write request, a core sends a write request and a cache line to the memory controller over a NoC column. For the purposes of our NoC scheduling policy, these two requests are analogous. We use a read request to demonstrate the calculation of C_M for a given core.

Figure 2 shows the progression of a memory read request initiated by the core farthest from the memory controller, namely core A, and the subsequent return of the requested memory line. Let us assume that the bandwidth of each link on the NoC is sufficiently large to transfer an entire request in one packet, but that a memory line requires four packets for its transfer (note that this is a realistic assumption on an architecture such as the TilePro64). The memory request packet then takes 4 cycles to reach the memory controller from core A. Let us assume that the latency of processing the memory request and actually receiving the requested line from memory is x cycles. So, the requested line arrives at the memory controller at cycle $4+x$ relative to the start of the request, reaches core D at cycle $5+x$ and so on. Subsequent packets arrive in successive cycles. Following this approach, the first packet, $P1$, reaches its final destination, core A, at time $8+x$ and the final packet, $P4$, reaches core A at time $11+x$. Among these cycles, we consider only the on-chip latency (i.e., the portion of the latency using the NoC). Hence, C_M^A is 11 cycles. Similarly, C_M^B is 9, C_M^C is 7 and C_M^D is 5. In summary, C_M^k for a given core k depends on the distance of the core, in terms of number of hops over the NoC column, from the memory controller designated to serve it.

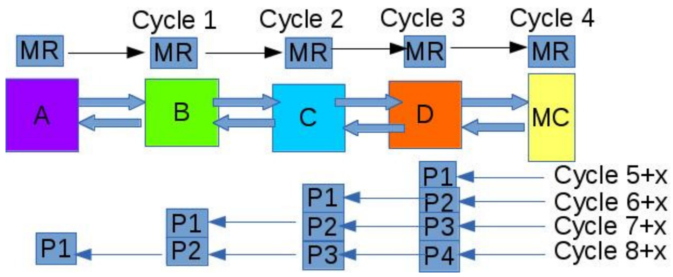


Fig. 2. Calculating C_M^D

Formally, C_M is calculated as shown in Equations 1 and 2 for read and write requests, respectively. Here, h^k is the number of hops between core k and the memory controller, r is the size of a request, l is the size of a cache line and b is the bandwidth of each link. For a read request, the first term represents the time for the request to travel over the NoC and the second term represents the time for the returned cache line to travel over the NoC. In the case of a write request, the request and cache line are both sent over the NoC one after another.

$$C_M^k = (h^k + \lceil \frac{r}{b} \rceil - 1) + (h^k + \lceil \frac{l}{b} \rceil - 1) \tag{1}$$

$$C_M^k = (h^k + \lceil \frac{r+l}{b} \rceil - 1) \tag{2}$$

C. Location-aware Partitioning and Calculation of T_M

As mentioned in the previous section, C_M^k for a core k is proportional the core's distance from the memory controller. Hence, it is advantageous to allocate tasks with a higher number of memory requests to a core closer to the memory controller. In the context of our paper, since tasks are allowed to lock their memory footprints in the cache on the core to which they are allocated, only lines that they are unable to lock lead to off-chip memory accesses. Since we assume that a task's memory footprint does not exceed one cache way, the only reason for unlocking cache lines would be conflicts with other tasks allocated to the same core. Hence, the goal of our location-aware partitioning algorithm is to ensure that 1) cache conflicts are reduced to the extent possible, 2) when unlocking becomes necessary due to unavoidable conflicts, the choice of which task will unlock lines is made with the goal of improved schedulability and 3) cores containing tasks with a higher number of off-chip memory accesses are closer to the memory controller. Algorithm 1 shows the steps involved in our approach. We now explain the working of our algorithm and the calculation of T_M for cores with the help of a simple running example.

Algorithm 1 Task Allocation Algorithm

```
1:  Function: allocate_tasks()
2:  order_tasks_non_increasing_utilization($task\_list$)
3:  for each $task \in task\_list$ do
4:      $allocated \leftarrow$ try_allocate_fully_locked($task$)
5:      if ($allocated = false$) then
6:          $allocated \leftarrow$ try_allocate_partially_unlocked($task$)
7:          if ($allocated = false$) then
8:              {Task set is unschedulable}
9:              EXIT
10:         end if
11:     end if
12: end for
13:
14: Function: try_allocate_fully_locked($task$)
15: for each $core \in available\_cores$ do
16:     $prev\_util$ = find_util($core$)
17:     allocate_task_to_core($task$, $core$)
18:     $cur\_util$ = find_util($core$)
19:     if ($cur\_util \leq 1$) then
20:         if ($cur\_util - prev\_util$ = locked_util($task$)) then
21:             return (true)
22:         end if
23:     else
24:         deallocate_task_from_core($task$, $core$)
25:         return (false)
26:     end if
27: end for
28:
29: Function: try_allocate_partially_unlocked($task$)
30: for each $core \in available\_cores$ do
31:     $prev$ = find_period($core$)
32:     allocate_task_to_core($task$, $core$)
33:     $schedulable \leftarrow$ check_schedulability($core$)
34:     {Use Equations 4, 3 and 5 for schedulability test}
35:     if $schedulable$ then
36:         $curr$ = find_period($core$)
37:         $change$ = $prev$ - $curr$
38:         add_info($sched\_core\_info\_list$, $core$, $change$)
39:         deallocate_task_from_core($task$, $core$)
40:     end if
41: end for
42: if is_empty($schedulable\_core\_info\_list$) then
43:     return (false)
44: else
45:     $core \leftarrow$min_period_change_core($sched\_core\_info\_list$)
46:     allocate_task_to_core($task$, $core$)
47:     relocate_cores_on_noc()
48:     {Relocate such that cores with shorter periods are closer
        to memory controller}
49:     return (true)
50: end if
```

1) Running Example Setup: Table I lists the characteristics of the task set used as a running example. The first, second and third columns represent the ID, period and base WCET of tasks, respectively. Recall that the base WCET is calculated assuming a task's entire memory footprint is locked in cache. The fourth column represents the number of accesses a given task performs to its locked cache footprint, termed its access frequency. Tasks $1-8$ have the same characteristics, so they are consolidated into a single row. Figure 3 shows the architectural

i	T_i	C_i	AF_i
1-8	100000	25000	3000
9	100000	25000	294
10	100000	25000	347

TABLE I. Task Set Characteristics for Running Example

setup used for our running example. The setup consists of 4 cores organized in a single column that share a port within a memory controller, i.e., the setup is essentially one column of the NoC shown in Figure 1. Each core has a private, lockable 2-way set associative cache. For the sake of illustration, in our running example, we assume that the memory footprints of all tasks map to the same cache sets. This implies that at most two tasks can lock their contents in any given set without conflicting.

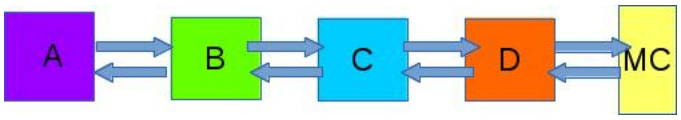


Fig. 3. Architectural Setup for Running Example

2) *Task Ordering:* The first stage in task partitioning is determining in what order tasks are chosen for allocation. In our algorithm, tasks are chosen for allocation in non-increasing order of their base utilizations (Line 2 in Algorithm 1). In our running example, since all tasks have the same base utilization, we choose them in order of their index or ID.

3) *Task-to-Core Allocation:* The second stage in task partitioning is determining where to allocate a chosen task. In our algorithm, there are two factors to consider when allocating a given task to a core. The first factor is schedulability of tasks on a given core. The second factor is schedulability of core memory requests, which are abstracted using task τ_M^k on each core k, on the NoC. Since we employ an EDF policy for scheduling tasks on cores and for scheduling abstract tasks (i.e., memory requests) on the NoC, schedulability conditions are simply based on utilization. The tests for core schedulability and NoC schedulability are shown in Inequalities 3 and 4, respectively. Here, $Util_i^k$ denotes the utilization of task τ_i on core k and m is the number of cores sharing a memory controller port (in our case, $m = 4$).

$$\sum_{\tau_i \in k} Util_i^k \leq 1 \tag{3}$$

$$\sum_{k=1}^{m} \frac{C_M^k}{T_M^k} \leq 1 \tag{4}$$

We now describe the steps involved in the task allocation process, the calculation of task utilization on a given core and the calculation of the periods of abstract tasks for cores.

a) *Allocation without Unlocking:* Our algorithm first attempts to allocate a chosen task, say τ_i, to a core where its base utilization can be accommodated and where it does not suffer any cache conflicts with other tasks (Function $try\_allocate\_fully\_locked()$ in Algorithm 1). In other words, it attempts to allocate task τ_i to a core, say k, where τ_i it is able to lock its entire memory footprint, thus retaining its base utilization (i.e., $Util_i^k = Util_i$). If more than one such core exists, the one with least current utilization is chosen in an effort to balance load better. In our running example, since we assume a 2-way set associative cache, at most two tasks with overlapping cache footprints can be accommodated on each core without any actual cache conflicts. Tasks τ_1 through τ_8 are found to be schedulable in this step on cores A through D. Note that, since these tasks are able to lock their entire footprints on their respective cores, these allocations affect only core schedulability and not NoC schedulability. The state of the system after this step is shown in Figure 4.

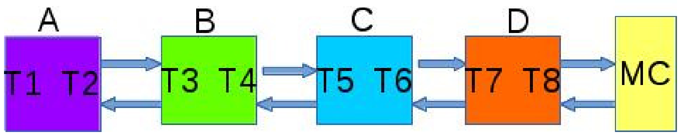


Fig. 4. System After Allocation of Tasks τ_1 through τ_8

b) *Allocation with Partial Unlocking:* If a chosen task τ_i cannot be accommodated on any core with its entire footprint locked, allocation with unlocking of some cache lines is explored (Function $try\_allocate\_partially\_unlocked()$ in Algorithm 1). Since unlocking of cache lines results in requests to main memory, NoC schedulability is affected in addition to core schedulability due to such an allocation. So, the goal of this step is to find a core where τ_i can be accommodated and both the core and the NoC remain schedulable. The algorithm proceeds to the steps described next.

i) *Slack-aware Unlocking*: A core, say k, is chosen from cores that have not previously been considered for the allocation of task τ_i, starting from the core that is closest to the memory controller. If core k cannot even accommodate the base utilization of task τ_i, it is not considered further and the algorithm skips to step iii). Otherwise, the algorithm proceeds as follows. If the allocation of τ_i on core k would result in a cache conflict on core k, either task τ_i must unlock (a subset of) its lines or an existing task on core k must do the same. Here, we employ a policy based on slack and access frequency. Slack of a task is defined as the difference between the task's period and its base WCET. Among the tasks with a cache conflict, the task that has highest ratio of slack to access frequency of the chunk to be unlocked is chosen for unlocking. The goal of this heuristic is to minimize the increase in utilization on the core and the increase in memory traffic over the NoC. Equation 5 shows the calculation of updated utilization for task τ_i and the same applies to any task on core k that ends up unlocking (a subset of) its cache lines. Here, AF_i^k is the access frequency to the chunk of lines that task τ_i ends up unlocking on core k.

$$Util_i^k = Util_i + \frac{AF_i^k \times [T_M^k]}{T_i} \tag{5}$$

The only unknown value in Equation 5 is T_M^k, which is calculated as shown next.

ii) *Calculation of T_M^k and Schedulability Check*: If core k has no unlocked cache lines at all, $T_M^k = 0$ since the core initiates no off-chip memory requests. If core k had unlocked cache lines before the consideration of task τ_i, T_M^k would already have a value assigned in a previous iteration. If the allocation of task τ_i on core k does not result in any *changes* to cache line unlocking, T_M^k remains unchanged. If the allocation of τ_i would result in a change to cache line unlocking on core k in accordance with step i), T_M^k must be updated. As mentioned earlier, T_M^k is the minimum inter-arrival time between memory requests from a core k. The value of T_M^k must be chosen such that it maintains schedulability of tasks on core k and schedulability of memory accesses on the NoC.

Inequality 3 represents the schedulability of tasks on core k. This imposes an upper bound on the inter-arrival time between memory requests for core k, i.e., an upper bound on the period, namely T_M^k, of the abstract task on core k. We set the value of T_M^k to this upper bound and use the updated characteristics of the abstract task in Inequality 4 to check for schedulability of memory requests on the NoC. If Inequality 4 is satisfied, task τ_i is schedulable on core k, i.e., core k is considered a *candidate* core for the allocation of task τ_i.

iii) Choose Best Candidate Core (Line 45 in Algorithm 1): Steps i) and ii) are repeated for every core. If no candidate core is found, the task set is declared unschedulable and the algorithm terminates. If, on the other hand, multiple candidate cores are identified, the core that leads to minimum increase in the NoC utilization is chosen. If there are multiple such cores, the core that suffers the minimum increase in its own utilization is chosen. Finally, if there are multiple such cores, the core with the lowest absolute utilization is chosen.

iv) Relocation of Cores (Line 47 in Algorithm 1): As mentioned above, the lower the value of T_M^k, the more frequent the number of memory accesses core k issues. In order to reduce the NoC utilization and improve NoC schedulability, the core with the lowest value of T_M should, in general, be closest to the memory controller since it would then have lowest values of C_M. As observed in earlier steps, when a new task τ_i is allocated to core k, it is possible that T_M^k decreases due to cache line unlocking. In this process, it is possible that T_M^k becomes less than the value of T_M^l, where core l is closer to the memory controller. In this situation, our algorithm (virtually) relocates core k to the position of core l and moves all cores starting from core l one position further away from the memory controller. We now prove that such relocation of cores is guaranteed not to jeopardize schedulability.

First of all, since we only relocate cores, but do not change the tasks allocated to cores or the periods of the abstract tasks on cores, the utilization of tasks on relocated cores, calculated according to Equation 5, and hence, the schedulability of the cores, remain unchanged. So, we only need to prove that the schedulability of the NoC is not jeopardized by relocation. Let $C1$ denote the on-chip memory access latency for core l before relocation and let $C2$ denote that of core k before relocation. By definition of on-chip memory access latency, $C1 < C2$. The combined contribution of cores l and k to NoC utilization before and after relocation are given by Equations 6 and 7, respectively. In both equations, the first term is the NoC utilization of l and the second, that of k.

$$Util^{before} = \frac{C1}{T_M^l} + \frac{C2}{T_M^k} \tag{6}$$

$$Util^{after} = \frac{C2}{T_M^l} + \frac{C1}{T_M^k} \tag{7}$$

Subtracting Equation 6 from Equation 6 and simplifying, we get Equation 8.

$$Util^{before} - Util^{after} = \frac{(C2 - C1)(T_M^l - T_M^k)}{T_M^l * T_M^k} \tag{8}$$

Since $C1 < C2$ and $T_M^l > T_M^k$, Equation 8 evaluates to a positive value. In other words, the NoC utilization can only *decrease* due to the relocation. The argument used for core l can be repeated for all other cores that are relocated to positions further away from the memory controller than they were.

c) Running Example - Allocation with Partial Unlocking: In our running example, tasks τ_9 and τ_{10} fall into the category of tasks that must be allocated with partial unlocking. Among these two tasks, τ_9 is chosen for allocation first and then τ_{10}.

Allocation of τ_9:

i) Since core D is closest to the memory controller, we first try to allocate task τ_9 on core D. Existing tasks on core D are τ_7 and τ_8. Since all tasks have the same slack and τ_9 has the lowest access frequency among the three tasks, it has the highest ratio of slack to access frequency and it must unlock its cache lines.

ii) We evaluate Inequality 3 to calculate an upper bound for T_M^D that maintains the schedulability of core D. This evaluation is shown in Inequality 9.

$$\frac{25000}{100000} + \frac{25000}{100000} + \frac{25000}{100000} + 294 \times \frac{T_M^D}{100000} \leq 1 \tag{9}$$

From this inequality, we obtain $T_M^D = 85$. We now use this value in Inequality 4 to check for NoC schedulability, as shown in Inequality 10.

$$\frac{5}{85} \leq 1 \tag{10}$$

Since both core and NoC schedulability are satisfied, D is a candidate core for task τ_9.

iii) In a similar way, cores C through A are also found to be candidates for task τ_9. We find that allocating task τ_9 on core D results in the minimum increase in NoC utilization. Hence, task τ_9 is allocated on core D. Figure 5 denotes the state of the system after allocation of task τ_9.

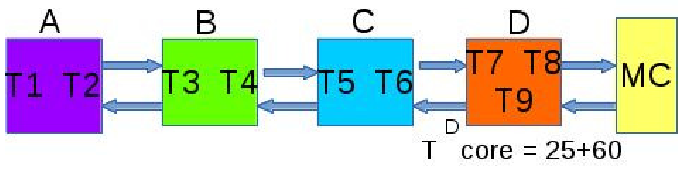


Fig. 5. State of system after allocation of T_9 Please Update the figure

iv) After the allocation of task τ_9, core D is the only core with unlocked cache lines. Since it is already closest to the memory controller, no core relocation is required.

Allocation of Task τ_{10}:

i) Core D is not considered for task τ_{10} since core D cannot accommodate even the task's base utilization. Next, our algorithm checks core C. On this core, it is found that τ_{10} would have to unlock its cache lines.

ii) Inequality 3 is evaluated to calculate an upper bound for T_M^D, as shown in Inequality 11.

$$\frac{25000}{100000} + \frac{25000}{100000} + \frac{25000}{100000} + 347 \times \frac{T_M^C}{100000} \leq 1 \quad (11)$$

From this inequality, we obtain $T_M^D = 72$. We now evaluate NoC schedulability, as shown in Inequality 12.

$$\frac{5}{85} + \frac{7}{72} \leq 1 \quad (12)$$

Since both core and NoC schedulability are satisfied, C is a candidate core for task τ_{10}.

iii) Core C is found to be the best candidate core for τ_{10} and the allocation is performed.

iv) Since core $T_M^C < T_M^D$ and core D is closer to the memory controller, we now perform a relocation of cores. Figure 6 shows the state of the system before and after the relocation of cores necessitated by the allocation of task τ_{10}.

D. Practical Considerations

We now provide a brief discussion on practical aspects involved in implementing EDF-on-NoC on a real platform. Although we assume a mesh-based NoC in this paper, it should be noted that the EDF-on-NoC policy itself can be applied to other topologies. However, the calculation of C_M and T_M may need to be updated according to the topology.

First, every core must have a buffer to hold outstanding memory requests from tasks. A given task busy waits after it performs a memory request. However, it is possible that the task gets preempted while it is waiting for a memory request to complete and the preempting task may perform another memory request. Hence, at any given time, there may be outstanding memory requests equal to the number of tasks that have unlocked cache lines on a given core. This number is known offline and is typically small. Since tasks performing memory requests can get preempted, every core must also have a buffer to hold returning memory lines until the requesting tasks resume execution and use them. Once again, the number of returned memory lines that need to be buffered is bounded by the number of tasks that have unlocked cache lines.

Note that a given task performs one memory request at a time. Hence, a task that has an outstanding memory request cannot also have unused returned data. Hence, a single buffer can be used for both purposes. An outstanding read request needs space for just the request and an outstanding write request needs space for the request as well as the cache line to be written. The size of the return data for a read request is equal to that memory line and that for a write request is empty. Hence, the size of the buffer is bounded by the number of tasks with unlocked cache lines times the size of data for a read request.

Second, the router on every core to be able to determine which packet to forward in any given cycle. To do so, every router must contain a lookup table to store the relative deadline (also equal to the period) of the abstract tasks of the cores that contend for a given port of a memory controller. Absolute deadlines of the "jobs" of this abstract task can be maintained online in this lookup table. Every router must also have a buffer that is capable of storing one packet per contending core at any given time. The number of contending cores is statically bounded for a given hardware platform. For example, on the TilePro64 platform, this number is 4.

IV. Prior Work used for Comparison

In previous work, we proposed a cache-aware semi-partitioned scheduling scheme [13] for the same task and architectural model that we assume in the current paper. In that work, the underlying partitioning policy (which is the first stage in a semi-partitioned scheduling algorithm) is a cache-aware policy. We also propose a weighted Time Division Multiple Access (TDMA)-based approach to bound memory access latency. We now briefly describe these two aspects of our previous work.

A. Weighted TDMA approach

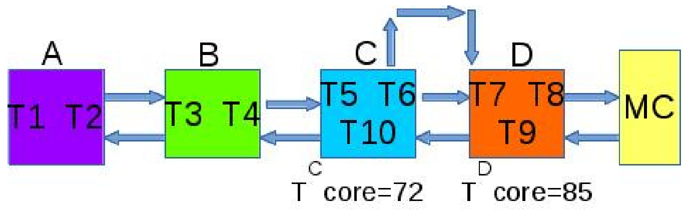
The basic idea of this scheme is to route all memory requests from a given core along a straight path to the appropriate memory controller and divide the available NoC bandwidth in inverse proportion to the core's distance, in terms of number of hops, from the memory controller. This scheme makes a task's WCET *independent* of the physical location of the core on which it executes. For the architectural setup shown in Figure 3, this weighted TDMA approach results in a memory access latency of $55 + 60$ cycles, where 55 is the latency over the NoC and 60 is the off-chip portion of memory access latency.

B. Cache-aware Partitioning

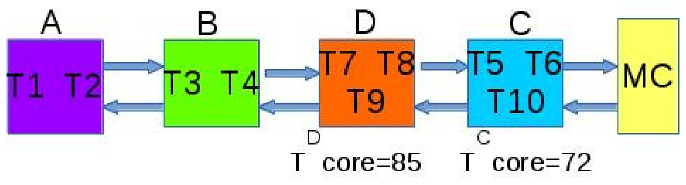
In this approach, tasks are chosen for allocation in non-increasing order of task utilization. Task allocation rules are as follows. Cores on which a chosen task's base utilization can be accommodated are identified and one core is chosen using a worst-fit strategy. If the chosen task is allocated to a core that already contains tasks, its locked cache regions may conflict with those of tasks already on the core. Thus, one or more tasks (including the new task) may be required to unlock a subset of their cache lines to resolve the conflicts. In this approach, the task that has the *minimum access frequency* to conflicting cache lines must unlock its cache lines. Unlocked cache lines result in memory accesses, thus requiring the use of the location-independent memory access latency calculated as described in the previous section. This increases the utilizations of the tasks on the core under consideration. If, after this step, there are multiple feasible cores, the algorithm chooses the core that suffers the *minimum change in utilization* due to the addition of the new task. If two or more cores have the same change in utilization, the core with the *minimum utilization* among them is chosen.

C. Running Example - Cache-aware Partitioning

Now, we apply the cache-aware partitioning scheme proposed in our previous work to our running example whose characteristics were shown in Table I. As is the case with our EDF-on-NoC approach, tasks τ_1 through τ_8 are allocated to cores A through D, as shown in Figure 4. However, task τ_9



(a) Before Relocation of cores



(b) After Relocation of cores

Fig. 6. State of system before and after reallocation of cores

is not able to get scheduled on any of the given cores. This is because of the pessimistic (location-independent) memory access latency employed in this approach. Equation 13 shows the calculation of the total utilization after the allocation of τ_9 for any of the cores (A to D).

$$Util = 0.75 + 294 \times \frac{[55 + 60]}{100000} \tag{13}$$

Here, $Util$ exceeds 1, making task τ_9 and, hence, the task set unschedulable.

V. Experimental Setup

We have implemented a software simulator for our algorithm. The architectural configuration used in our simulations is shown in Table II, where the 4 cores are organized as shown in Figure 3. The purpose of this setup is to demonstrate the working of our algorithm on a set of cores that share a given memory controller port. Note that our algorithm is not restricted to any specific number of cores sharing a memory controller port. The results of the demonstration can be directly extrapolated to a platform with several such NoC columns since there is no interference across such columns. On each core, we assume that all four ways of the L1 data cache are lockable.

Parameter	**Configuration**
Processor Model	in-order
Cache Line Size	32Bytes
L1 D-Cache Size/Associativity L1 hit latency	256KB/4-way 1 cycle
Number of Cores	4
Cache to cache Transfer latency	13 cycles
External Memory Latency	60 cycles

TABLE II. System Configuration

We generate and use synthetic task sets in order to evaluate our algorithm and compare its performance to our prior work [13]. The synthetic task sets are generated using an unbiased random task set generator that is based on an approach proposed by Bini et al. [28]. In order to impose the maximum stress on caches, we generate task sets whose cache conflict graph forms a clique, i.e., all tasks conflict at least partially with all other tasks. The memory footprint of a given task is divided into chunks. Each chunk has a given number of elements and a known access frequency that is proportional to the number of elements in the chunk. When a task must unlock lines, it must do so in the granularity of these chunks, i.e., a given chunk is either completely locked or completely unlocked.

VI. Experimental Results

In this section, we present the results of our evaluation of our prior scheme [13] and the scheme proposed in the current paper. Each of these schemes consists of three aspects, namely the policy used for scheduling memory requests on the NoC, the policy used for task allocation and, within that, the policy used to determine which task must unlock cache lines in the presence of cache conflicts during allocation. The NoC scheduling scheme of our prior work and current work are referred to as TDMA and EDF-on-NoC, respectively. The task allocation schemes in our prior work and current work are referred to as cache-aware partitioning or CAP and location-aware partitioning or LAP, respectively. Finally, the heuristics used to determine which task unlocks cache lines in prior work and current work are referred to as minimum access frequency or MAF and maximum slack ratio or MSR, respectively.

We present results from four different experiments. In each experiment, we build and compare two scheduling schemes, A and B, using a combination of three components, namely NoC scheduling policy, task allocation policy and cache line unlocking policy. We first randomly generate a set of 30 tasks with a total utilization of 4.[3] This set of 30 tasks is given to algorithm A, which attempts to schedule as many tasks as possible from the set of 30 tasks. The subset of task sets scheduled by A is now given as input to algorithm B and the results of the two are compared. This comparison is referred to as $A \hookrightarrow B$. Next, the same initial set of 30 tasks is taken. This time, the set is first given to algorithm B, which attempts to schedule as many tasks from the set as it can. The subset of tasks that algorithm B manages to schedule is then given as input to algorithm A. This comparison is referred to as $B \hookrightarrow A$. *This way, we are able to fairly compare the performance of the algorithms when they are each given a common input task set that at least one of them is able to feasibly schedule.* This process is repeated for 100 different task sets of 30 tasks each.

A. Experiment 1: Full Algorithm Comparison

In this experiment, we compare the performance of our complete prior scheme, i.e., algorithm A uses TDMA + CAP + MAF, and the complete scheme proposed in the current paper, i.e., algorithm B uses EDF-on-NoC + LAP + MSR. Figure

[3]Since we use a 4-core setup, the maximum utilization supported is 4.

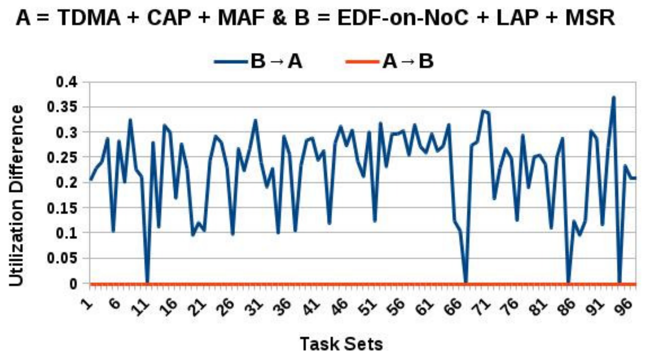


Fig. 7. Full Scheme Comparison

7 shows the result of this comparison. The x-axis shows task sets and the y-axis shows the difference in base utilizations scheduled by the two algorithms for the two cases, namely $A \hookrightarrow B$ and $B \hookrightarrow A$. From this figure, we observe that, in all cases, the difference in base utilization scheduled by the two algorithms is 0 for $A \hookrightarrow B$, indicating that task sets that were schedulable by A are also schedulable by B. In other words, task sets schedulable using the algorithm proposed in our prior work are always schedulable by the scheme proposed in the current paper. On the other hand, in the case of $B \hookrightarrow A$, we observe that this is not the case. 96% task sets exhibit a difference in scheduled utilization, i.e., most task sets that are schedulable by algorithm B (EDF-on-NoC + LAP + MSR) are *not schedulable* by algorithm A (TDMA + CAP + MAF).

B. Experiment 2 : Using Cache Unlocking Policy MAF

In this experiment, we employ a common cache line unlocking policy for both algorithm A and algorithm B, namely the minimum access frequency or MAF policy, which was used in our prior work [13]. In other words, algorithm A is built using TDMA + CAP + MAF and algorithm B is built using EDF-on-NoC + LAP + MAF. Figure 8 shows the results of

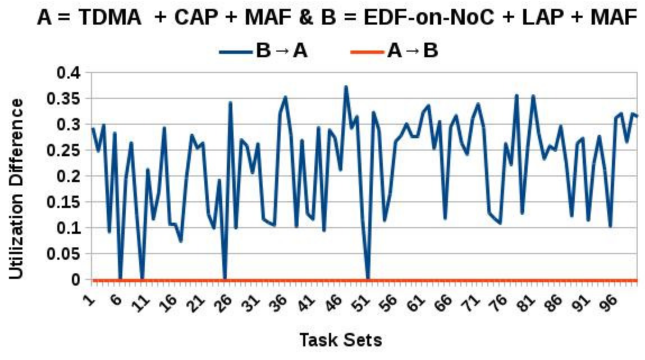


Fig. 8. Comparison with MAF cache line unlocking policy

this comparison. The format of the graph is the same as that in the previous experiment. Since a common cache unlocking policy is used, these results essentially compare TDMA + CAP and EDF-on-NoC + LAP. We observe from Figure 8 that all task sets schedulable using TDMA + CAP are also schedulable using EDF-on-NoC + LAP. However, 96% task sets that are schedulable by EDF-on-NoC + LAP are not schedulable by TDMA + CAP. This demonstrates that a combination of a less pessimistic NoC scheduling scheme and location awareness improves overall schedulability.

C. Experiment 3 : Using Cache Unlocking Policy MSR

In this experiment, once again, we employ a common cache line unlocking policy for both algorithm A and algorithm B, but the maximum slack ratio or MSR policy proposed in our current work. So, algorithm A is built using TDMA + CAP + MSR and algorithm B is built using EDF-on-NoC + LAP + MSR. Figure 9 shows the performance of the two algorithms.

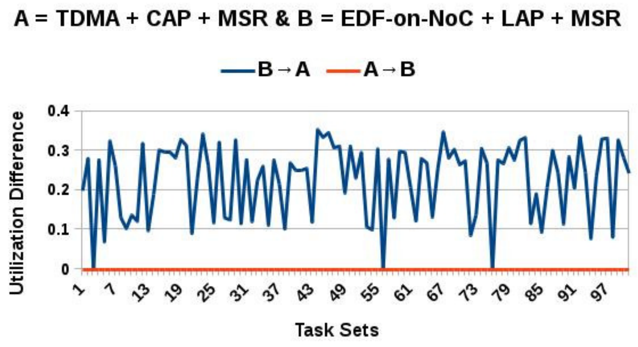


Fig. 9. Comparison with MSR cache line unlocking policy

From this figure, we observe once again that all task sets that are schedulable by TDMA + CAP are also schedulable by EDF-on-NoC + LAP, but 97% task sets that are schedulable by EDF-on-NoC + LAP are not schedulable by TDMA + CAP. We observe no specific pattern due to the difference in the cache line unlocking policies used in Experiments 2 and 3.

Overall, the results from these three experiments demonstrate that combination of location awareness and improved NoC scheduling affect task set schedulability significantly.

D. Experiment 4: Comparing TDMA and EDF-on-NoC

In the last experiment, we wish to compare the performance of the NoC scheduling policies, namely TDMA and EDF-on-NoC. In order to perform a fair comparison among the two policies, we employ a common underlying task allocation and cache line unlocking scheme. Specifically, we employ the cache-aware partitioning scheme or CAP along with minimum access frequency or MAF unlocking policy, as proposed in prior work [13]. The reason for this is that our location-aware partitioning is unsuitable for use with a location oblivious NoC scheduling scheme. So, essentially, algorithm A is TDMA + CAP + MAF and algorithm B is EDF-on-NoC + CAP + MAF.

Since a common underlying task allocation is employed, the base utilizations scheduled by algorithms A and B are the same. Our goal is to compare the effectiveness of NoC scheduling schemes. Hence, total utilization of the system under each algorithm is a more suitable metric for comparison. The total utilization is the sum of base utilization and the increase in utilization due to cache line unlocking. Figure 10 shows the result of this comparison. This figure shows the base utilization scheduled by both algorithms A and B and also the total utilizations obtained for each individual algorithm. We

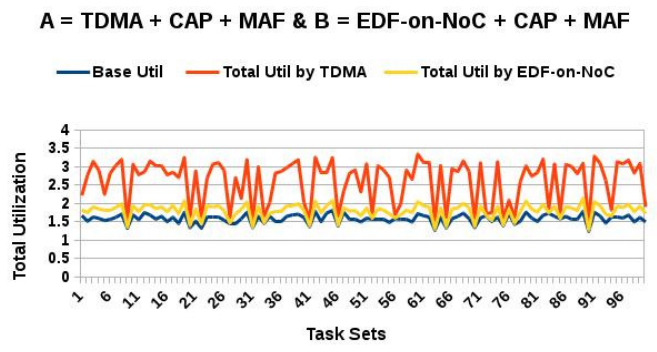


Fig. 10. Comparing EDF-on-NoC and Weighted TDMA policy

observe that the total utilization under the EDF-on-NoC policy is significantly less compared to that under the TDMA policy, with TDMA resulting in an average increase in utilization of 25%. This demonstrates the benefits of a less pessimistic NoC scheduling policy such as EDF-on-NoC.

VII. Conclusions and Future Work

In this paper, we present a dynamic-priority policy for scheduling memory requests on the Network-on-Chip of a cache-based many core platform. We use this policy to develop a location-aware partitioning algorithm for hard-real-time tasks that improves task set schedulability. We compare the performance of our proposed approach with a cache-aware partitioning scheme proposed in prior work and demonstrate that it achieves significantly improved performance.

In future work, we propose to conduct sensitivity studies with varying task set utilizations and varying cache usage patterns to further analyze our scheme. We also propose to compare the effectiveness of our scheme with other recent schemes for bounding response times of tasks with explicit consideration of interference over the Network-on-Chip. Finally, we propose to implement our schemes on a real hardware platform and study the practical applicability of the approach.

References

[1] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, 2011.

[2] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems." *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[3] J. Yan and W. Zhang, "Wcet analysis of multi-core processors with shared l2 instruction caches," in *IEEE Real-Time Embedded Technology and Applications Symposium*, Apr. 2008.

[4] S. Chattopadhyay, A. Roychoudhury, and T. Mitra, "Modeling shared cache and bus in multi-cores for timing analysis," in *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*, 2010, pp. 6:1–6:10.

[5] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-aware multicore wcet analysis through tdma offset bounds." in *ECRTS*, 2011, pp. 3–12.

[6] "Intel's single-chip cloud computer," http://techresearch.intel.com/ProjectDetails.aspx?Id=1.

[7] "Tilera processor family," http://www.tilera.com/.

[8] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip." *CSUR*, 2006.

[9] K. Goossens, J. Dielissen, and A. Radulescu, "thereal network on chip: concepts, architectures, and implementations," *IEEE Des Test*, vol. 22, no. 5, pp. 414–421, 2005.

[10] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in tempo- rally disjoint networks within the nostrum network on chip." in *Proceedings of the design automation and test Europe conference (DATE)*, 2004.

[11] P. Wolkotte, G. Smit, G. Rauwerda, and L. Smit, "An energy-efficient reconfigurable circuit- switched network-on-chip." in *Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS05)Workshop 3*, 2005.

[12] D. Wiklund and D. Liu, "Socbus: switched network on chip for hard real time embedded systems." in *Proceedings of the 17th international symposium on parallel and distributed processing*, 2003.

[13] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, "Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems," in *Euromicro Conference on Real-Time Systems*, Jul. 2012.

[14] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "Qnoc: Qos architecture and design process for network on chip." *J Syst Archit*, pp. 105–128, 2004.

[15] N. Kavaldjiev, G. Smith, P. Jansen, and P. Wolkotte, "A virtual channel network-on-chip for gt and be traffic." in *Proceedings of the IEEE computer society annual symposium on emerging VLSI technologies and architectures*, 2006.

[16] T. Bjerregaard and J. Sparso, "A 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*, 2005.

[17] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching." in *Proceeding of the 2nd ACM/IEEE international symposium on networks-on-chip (NoCS)*, 2008.

[18] ——, "Real-time communication analysis with a priority share policy in on-chip networks." in *ECRTS*, 2009.

[19] ——, "Schedulability analysis and task mapping for real-time on-chip communication," *Real Time Systems*, vol. 46, pp. 360–385, 2010.

[20] B. Nikolic, P. M. Yomsi, and S. M. Petters, "Worst-case memory traffic analysis for many-cores using a limited migrative model." in *RTCSA*, 2013.

[21] A. Sarkar, F. Mueller, and H. Ramaprasad, "Static task partitioning for locked caches in multi-core real-time systems," in *Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2012.

[22] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *In IEEE Real-Time Systems Symposium*, 2002, pp. 114–123.

[23] I. Puaut, "Wcet-centric software-controlled instruction caches for hard real-time systems," in *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 217–226.

[24] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Design, Automation and Test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 1484–1489. [Online]. Available: http://portal.acm.org/citation.cfm?id=1266366.1266692

[25] B. Lisper and X. Vera, "Data cache locking for higher program predictability," in *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Mar. 06 2003, pp. 272–282.

[26] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Design Automation Conference*. New York, NY, USA: ACM, 2008, pp. 300–303. [Online]. Available: http://portal.acm.org/citation.cfm?id=1391469.1391545

[27] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan, "Push-assisted migration of real-time tasks in multi-core processors," in *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, Jun. 2009, pp. 80–89.

[28] E. Bini and G.C.Buttazzo, "Measuring the performance of schedulability tests," *RTS*, vol. 30, no. 2, pp. 129–154, 2005.