Noname manuscript No.
(will be inserted by the editor)

Architecture Aware Semi Partitioned Real-Time Scheduling on Multicore Platforms

Mayank Shekh · Harini Ramaprasad · Abhik Sarkar · Frank Mueller

the date of receipt and acceptance should be inserted later

Abstract As real-time embedded systems integrate more and more functionality, they are demanding increasing amounts of computational power that can only be met by deploying them on powerful and scalable multicore architectures. The use of multicore architectures with on-chip memory hierarchies and shared communication infrastructure in the context of real-time systems poses several challenges for task scheduling.

Semi-partitioned scheduling algorithms form a middle ground between the two extreme approaches, namely global and partitioned scheduling. In such an algorithm, as many tasks as possible are partitioned onto cores and the remaining tasks are allowed to migrate in a pre-specified manner. By making most tasks non-migrating (partitioned), runtime migration overhead is minimized. On the other hand, by allowing some tasks to migrate among cores, schedulability is improved.

In this paper, we present a predictable semi-partitioned scheduling algorithm for independent hard-real-time sporadic tasks executing on homogeneous multicore

This work was supported in part by NSF grants CNS-0905212, CNS-0905181, and CNS-1239246. Author's addresses: Mayank Shekhar and Harini Ramaprasad, Department of Electrical and Computer Engineering, Southern Illinois University Carbondale; Abhik Sarkar (current address), Intel Corporation, California; Frank Mueller, Department of Computer Science, North Carolina State University. A preliminary version of this work appeared in the Euromicro Conference on Real-Time Systems (ECRTS), 2012 Shekhar et al (2012). This journal version motivates and proposes a novel task ordering scheme based on task migration characteristics (Section 5.3.1) and a novel Slack and Architecture Aware task allocation scheme (Sections 5.3.4 and 5.3.5). It presents simulation results for the newly proposed schemes and performs a detailed comparison between the new schemes and two existing schemes (Section 8).

M. Shekhar
Southern Illinois University Carbondale

H. Ramaprasad
Southern Illinois University Carbondale

A. Sarkar
North Carolina State University

F. Mueller
North Carolina State University

platforms using cache locking and locked cache migration. As part of the semi-partitioned scheduling algorithm, we propose two different task ordering schemes and two different schemes for the initial partitioning phase. Simulation results demonstrate the effectiveness of the proposed schemes in comparison to existing state-of-the-art techniques.

1 Introduction

As real-time embedded systems integrate more and more functionality, they are demanding increasing amounts of computational power that can only be met by using multicore architectures. Modern scalable multicore architectures typically have on-chip memory hierarchies (e.g., caches) and other shared on-chip resources such as the communication infrastructure. Since real-time systems require *a-priori* guarantees of task set schedulability, the use of such multicore architectures in these systems poses several challenges. It is imperative that tasks are carefully scheduled and accesses to shared on-chip resources are suitably arbitrated to guarantee functional and timing correctness without severely compromising schedulable utilization or introducing unreasonable imbalance in core loads.

Multicore real-time scheduling algorithms may be broadly classified into two categories, namely global and partitioned algorithms. In global scheduling (Baruah et al, 1996; Moir and Ramamurthy, 1999; Anderson and Srinivasan, 2000; Srinivasan and Anderson, 2002; Baruah, 2007), all jobs are stored in a single prioritized queue and the scheduler allocates them to cores according to their priority. As a result, jobs may be scheduled on different cores at different times, thus requiring migration of jobs among cores. While this allows for optimal scheduling policies, high schedulable utilization and good load balancing, ensuring predictable task migration is challenging.

On the other hand, in partitioned scheduling(Dhall and Liu, 1978; Burchard et al, 1995), tasks are statically partitioned onto cores and remain there throughout their lifetime. A local scheduler schedules tasks on a given core using some unicore scheduling algorithm. The advantage of this approach is improved predictability due to elimination of online migration overhead. However, deriving an optimal partitioning of tasks is a NP hard problem. Hence, schedulable utilizations that can be achieved in a partitioned approach are typically much lower than those in global scheduling algorithms and load imbalance may be unavoidable due to task set characteristics.

Between the two extremes, we have a third category that is called semi-partitioned scheduling (Kato and Yamasaki, 2008; Andersson and Bletsas, 2008; Dorin et al, 2010; Burns et al, 2010). Here, as many tasks as possible are statically partitioned onto cores while the rest are allowed to migrate among a fixed subset of cores. Recently, Anderson *et al.* (Anderson et al, 2011) analyzed the practicality of semi-partitioned scheduling. They explicitly included overheads derived from actual measurements in their experiments and concluded that it is a sound approach for both hard- and soft-real-time systems. In this paper, we present a semi-partitioned approach for predictably scheduling sporadic hard-real-time tasks using locked-cache migration on a Network-on-Chip (NoC) based multicore architecture.

A semi-partitioned scheduling scheme has two phases, namely the partitioning phase and the phase in which migrating tasks are allocated to cores. The overall goal of semi-partitioned scheduling is to statically partition as many tasks as possible onto cores in an effort to minimize online migration overhead, and allow the remaining tasks to migrate, typically in a predetermined manner among preselected cores. In this paper, we demonstrate the importance of a good underlying task partitioning algorithm in the overall success of a semi-partitioned scheduling scheme. Furthermore, we demonstrate that even a simple (potentially sub-optimal) algorithm for allocating migrating tasks, when used in conjunction with a good partitioning scheme, can result in good schedulable utilization.

The success of the first phase of any semi-partitioned scheduling scheme, namely task partitioning, is dependent on two factors. The first factor is the order in which tasks are chosen for allocation and the second is the policy used to determine which core to allocate the chosen task to. In this paper, we present two task ordering schemes and two task allocation schemes.

The first task allocation scheme allocates tasks using cache- and NoC-aware heuristics. Although the first task allocation scheme is shown to be efficient under the architectural assumptions made in this paper, the technique is greedy in the sense that it does not consider the effects of task partitioning on the allocation of migration tasks performed in the second phase. We show that such a migration-unaware partitioning scheme could result in decreased schedulable utilization. To improve schedulability in this context, we present a second task partitioning scheme that is migration-aware in addition to being architecture-aware.

In order to improve the predictability of multi-task execution on a single core, we allow tasks on a given core to *statically* choose and lock a subset of their memory lines in the core's private cache. For a migrating task, locked cache lines belonging to the task are proactively migrated with the task and re-locked on its target core. Since we use locked cache migration, migration overhead is predictable even though a task may be migrated in the middle of a given job's execution. In order to guarantee predictable sharing of the on-chip communication infrastructure, we employ a time-division multiplexed (TDM) arbitration scheme.

We demonstrate the effectiveness of our techniques through simulations on synthetic task sets generated using an unbiased random task set generator. In addition to comparing the proposed partitioning schemes and ordering schemes among each other, we compare our techniques with two existing semi-partitioned scheduling approaches (Burns et al, 2010; Kato et al, 2009).

2 Assumptions

Architectural and Task Model. We assume a homogeneous multicore architecture, where each core has private, set associative, lockable caches, and a two dimensional (2D) mesh-based NoC interconnect with dedicated, bidirectional channels for cache-to-cache transfers between cores that does not interfere with channels for regular main memory accesses. An example of such an architecture in practice is the recent 64-core TilePro64 architecture from Tilera (Tilera, 2012) that has five independent

mesh-based NoC interconnects[1]. We also assume support for message prioritization similar to that provided by the CAN bus protocol (Livani et al, 1999). We assume that each core's router has a buffer with size at least equal to the size of one cache line and that all memory requests are pipelined at the memory controller and that the memory access latency includes the delay due to pipelining.

We assume a sporadic hard-real-time task model where relative deadlines of tasks are less than or equal to their respective minimum inter-arrival times. A task T_i is represented by the tuple (P_i, C_i, D_i), where P_i is the minimum inter-arrival time of T_i, hereafter referred to as its period for simplicity, C_i is the worst-case execution time (WCET) and D_i is the relative deadline of T_i. The utilization of T_i, denoted as u_i, is calculated as the ratio of its WCET to its period, i.e., $u_i = C_i/P_i$. We assume that tasks are independent of each other and may lock cache lines on the core to which they are allocated. Memory lines that are not locked are assumed to bypass the cache. For each task, a static timing analyzer developed in prior work (Ramaprasad and Mueller, 2010) is used to calculate the worst-case execution time (WCET) of the task with a chosen set of cache lines locked (C_i^{locked}) [2] and its corresponding utilization is denoted as u_i^{locked}. We assume that partitioned tasks may use at most $k-1$ ways of the k available ways in a core's private cache, that the k^{th} way is dedicated for migrating tasks, and that only one migrating task may be allocated to any given core. The latter is done in order to avoid the possibility of contention among migration traffic that may arise if two or more migrating tasks are allocated to the same core and, thus, minimize migration overhead. In the current paper, we use the Earliest Deadline First (EDF) scheduling policy on each core.

NoC Routing and Arbitration Model. Memory requests issued by cores are assumed to be statically routed along a straight path to the memory controller and arbitrated using a time-division-multiplexed (TDM) approach, as described below. Consider the example mesh in Figure 1. We only depict the channel for memory traffic and consider the flow of traffic to/from a single memory controller M along a straight vertical path.

Bus C conveys traffic from cores A, B and C, Bus B that from cores A and B, and Bus A from core A. The bandwidth allocated to memory traffic from a given core along a given bus is proportional to the number of hops from the core to the target of that bus. For example, the bandwidth along Bus C is divided among cores A, B and C in the ratio 3:2:1 since traffic from core A crosses three hops to get to the target of Bus C (memory controller M), core B crosses two hops and core C crosses one hop. If we assume that transfer of a unit of data along each hop takes one cycle, the NoC latencies for cores A, B and C across Bus C are 2, 3 and 6 cycles, respectively. Similarly, NoC latencies for cores A and B across Bus B are 2 and 3 cycles and that for core A across Bus A is 1 cycle. Hence, the total NoC latency for traffic from core A is 5 cycles, that for traffic from core B 6 cycles, and that for traffic from core C 6

[1] Note that we do not claim that the TilePro64 architecture is a typical embedded architecture. In contrast, our aim is to make scalable architectures like the TilePro64 suitable for real-time task execution since they can provide the performance that is being demanded by modern real-time systems

[2] We assume that the regions that each task wishes to lock are pre-selected. Methods used to make this choice are out of the scope of this paper.

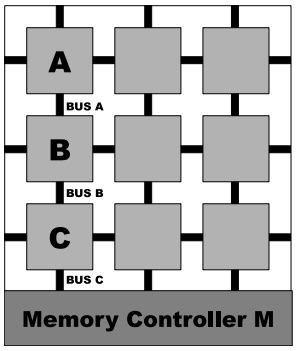

Fig. 1 Memory Traffic Routing

cycles. For safety, we assume that the NoC latency of every memory request is the maximum of these three latencies, namely 6 cycles.

By employing the prioritized TDM approach described above, the main memory access time for a given core becomes both predictable and independent of the physical location of the core on the mesh. In turn, this makes the worst-case execution time (WCET) of a task independent of the physical location of the core on which it is allocated. Although the use of such a TDM scheme may result in underutilization of the NoC bandwidth, we believe that it is an acceptable trade-off since it improves predictability. Location independence also enables us to perform task allocation onto virtual cores and then map virtual cores that contain different portions of a given migrating task physically close to each other in an effort to decrease online migration overhead.

Architectures such as Tilera's TilePro64 have main memory symmetrically distributed into four parts around the chip, each serviced by a separate memory controller. Since our latency calculation assumes that all traffic from a given core is routed along the straight path to the appropriate memory controller, it easily extends to this case.

3 Background and Related Work

In this section, we present relevant background information and discuss related work.

3.1 Cache Locking

Schedulability theory for real-time systems requires *a-priori* estimates on the worst-case execution times (WCETs) of tasks. Architectural features such as caches complicate the process of estimating task WCETs, specially in the context of prioritized multi-task systems. Cache locking is a technique that may be used to improve the timing predictability of real-time tasks. The idea is that a task may explicitly load and lock predetermined content into the cache. For the duration that the content is locked, cache behavior becomes completely predictable. Cache locks may be applied

statically or dynamically. In static cache locking, the system locks cache lines for a given task during the start-up phase and these lines remain locked during the lifetime of the task. On the other hand, dynamic locking allows a task to change the contents of its locked regions at pre-selected cache reload points. In our paper, we employ static cache locking.

Several techniques have been proposed in recent years for static and dynamic cache locking. Puaut *et al.* have proposed static and dynamic cache locking techniques for instruction caches (Puaut and Decotigny, 2002; Puaut, 2006) and cache locking techniques that provide performance comparable with scratchpad-based techniques (Puaut and Pais, 2007). Lisper *et al.* have proposed techniques for locking data caches (Lisper and Vera, 2003). Recently, cache locking techniques for multicore systems with shared L2 caches have been proposed by Suhendra *et al.* (Suhendra and Mitra, 2008).

3.2 Networks-on-Chip

A Network-On-Chip (NoC) is an infrastructure designed for communication among the resources on a chip as an alternative to traditional bus-based communication and has been extensively studied by researchers (Chou and Marculescu, 2008; Murali and Micheli, 2004; Rhee et al, 2004; van den Brand et al, 2007; Goossens et al, 2005; Shi and Burns, 2008, 2010) Every core has a router that makes decisions for movement of data through the NoC. Efficiency of communication depends on the topology of the network and the routing algorithm used. A two-dimensional mesh topology has been found to be a viable solution for many-core architectures, providing both massive bandwidth and scalability in practice (Tilera, 2012; Intel, 2012).

3.3 Real-Time Scheduling on Multicores

Scheduling of real-time tasks on cores is paramount to properly utilize multicore systems. Multicore scheduling schemes may be broadly classified into partitioned and global scheduling policies.

In *partitioned scheduling*, tasks are assigned to cores statically and are not allowed to migrate between cores (Dhall and Liu, 1978; Burchard et al, 1995). The advantage is that there is no migration overhead. However, partitioned schemes have three main disadvantages. First, they are inflexible and cannot easily accommodate dynamic tasks without a complete re-partition. The re-partitioning problem may be resolved by allocating incoming dynamic tasks to the first available core, but this may not be optimal in terms of overall system utilization. Second, optimal assignment of tasks to cores is an NP-hard problem for which polynomial-time solutions result in sub-optimal partitions, thereby resulting in lower schedulable utilizations. Finally, task set characteristics may force an unbalanced load on cores.

In *global scheduling* policies, tasks are allowed to migrate among cores. An advantage of task migration is that it may be used to dynamically balance the system load and allow optimal scheduling of tasks onto cores (Baruah et al, 1996; Moir

and Ramamurthy, 1999; Anderson and Srinivasan, 2000; Srinivasan and Anderson, 2002; Baruah, 2007). However, the number of task migrations introduced in such schemes could be prohibitive in the context of real-time systems due to online migration overheads that change the timing behavior of tasks, thus affecting the overall timing predictability of the system.

3.3.1 Semi-partitioned Scheduling

Between the two extremes of partitioned and global scheduling lies the idea of semi-partitioned scheduling. Bastoni, Brandenburg and Anderson recently conducted an experimental study exploring the practicality of semi-partitioned scheduling and found that it is indeed a feasible approach in general (Anderson et al, 2011). Several schemes for semi-partitioned scheduling have been proposed (Dorin et al, 2010; Kato et al, 2009; Burns et al, 2010). Semi-partitioned scheduling algorithms typically take one of two approaches. In the first approach, different jobs of a migrating task may be allocated to different cores, but a given job executes completely on a single core (Dorin et al, 2010). In the second approach, a task is split into parts, with each part being scheduled on a different core, and every job of a given task always executes on the same set of cores (Burns et al, 2010; Kato et al, 2009). In our paper, we employ the second approach.

Although existing semi-partitioned scheduling approaches significantly reduce the number of task migrations compared to global scheduling policies, they do not explicitly consider architectural features of multi-cores, including caches and communication infrastructure. Instead, they assume that constants may be added to task WCETs, given a bound on the number of migrations, to account for migration costs. In cache-based multi-core architectures with NoCs, calculation of migration overheads is not trivial. Furthermore, the effectiveness of a task allocation scheme may be significantly diminished if it does not explicitly consider such overheads as an integral part of the scheme itself.

In this paper, we introduce a semi-partitioned scheduling scheme that explicitly considers overheads due to cache content migration in the context of dedicated cache based multi-core architectures with a NoC interconnect. Sarkar *et al.* have proposed proactive, push-based migration mechanisms for bus-based multicore architectures (Sarkar et al, 2009) and mechanisms to support migration of locked cache lines among cores (Sarkar et al, 2011). In our current paper, we adopt such a push-based migration mechanism.

To evaluate the effectiveness of the techniques proposed in this paper, we compare them with two existing semi-partitioned scheduling schemes, namely a scheme proposed by Burns *et al.* (Burns et al, 2010) and a scheme proposed by Kato *et al.* (Kato et al, 2009). Since these schemes do not explicitly consider cache migration overheads, in order to enable a fair comparison, we integrate our calculation of cache content migration overhead into the two existing schemes and evaluate them alongside our schemes. Our reason for choosing these particular schemes for comparison is that they are both state-of-the-art approaches and they both use a task-splitting approach to semi-partitioned scheduling, as we do in the current paper. **We describe the two mentioned scheme in detail in the next section.**

4 Compared Semi-Partitioned Algorithms

In this section, we briefly explain the two algorithms Burns et al (2010); Kato et al (2009) which we are comparing against in this paper. We add the additional overhead to both the algorithms for the architectural constraints considered in this paper for a fair comparison.

4.1 Processor Demand Analysis

We now briefly describe processor demand analysis or PDA, which is a necessary and sufficient schedulability test for a system of tasks with relative deadlines less than their periods, scheduled on a single core using the Earliest Deadline First (EDF) policy. This test is used in two of the three semi-partitioned algorithms evaluated in this paper Burns et al (2010); Kato et al (2009).

The principle behind this test is that the demand on a processor should always be less than or equal to the processor's supply. Equation 1 represents the basic schedulability condition for PDA.

$$\forall t > 0, d(\tau^p, t) < t \tag{1}$$

Here, $d(p,t)$ is the demand imposed on a processor p by the set of tasks τ^p allocated to the processor, in an interval of time equal to t. It can be calculated as the sum of execution times of all jobs that have a deadline at or before t, as shown in Equation 2.

$$d(\tau^p, t) = \sum_{\forall i \in \tau^p} \lfloor (t + P_i - D_i)/P_i \rfloor C_i \tag{2}$$

Here, n is the number of tasks on a given processor. The schedulability of the task set is checked for all values of t in an interval $0 - L$, where an upper bound of the value of L is the hyperperiod of the task set. A tighter bound on L, denoted as L_B, is equal to the length of an in-phase busy interval of the task set Ripoll et al (1996). Equation 3 calculates this busy interval using a recurrence relation that stops when $busy\_interval(k+1)$ is equal to $busy\_interval(k)$.

$$busy\_interval(k+1) = \sum_{i=1}^{i=n} \lfloor busy\_interval(k)/T_i \rfloor C_i \tag{3}$$

If the total utilization is less than or equal 1, i.e., if the task set is schedulable, the recurrence in Equation 3 is guaranteed to terminate for some value of k. Then, L_B is equal to $busy\_interval(k)$.

When total utilization is strictly less than 1, a simpler bound on the value of L may been derived Hoang et al (2006). Equation 4 shows this calculation. Here, u_i is the utilization of task T_i and U indicates the total utilization of the task set.

$$L_A = Max[D_1, D_2, \ldots..D_n, \sum_{i=1}^{i=n} (P_i - D_i)u_i/(1-U)] \tag{4}$$

In order to reduce the computational overhead in PDA, another method called Quick convergence Processor Demand Analysis (QPA) was proposed Zhang and Burns (2008). In this method, we start with the upper bound of L and then iterate backwards until its value is less than the minimum deadline of the task set. At every iteration, the value of L is recalculated.

4.2 C=D Task Splitting Scheme

This section briefly describes the semi-partitioned scheduling algorithm proposed by Burns *et al.* Burns et al (2010). In this algorithm, migrating tasks are split into two parts such that the first part has a relative deadline equal to its WCET, hence giving the algorithm its name. Due to its characteristics, the first part of such a split task always executes at highest priority on the core to which it is assigned. The second part of the task executes as a regular task on the core to which it is assigned, with a phase equal to the WCET of the first part. Formally, task T_i (ϕ_i, C_i, D_i, P_i) is split into T_i^1 (ϕ_i, C_i^1, $D_i^1 = C_i^1$, P_i) and T_i^2 ($\phi_i + C_i^1$, $C_i^2 = (C_i - C_i^1)$, D_i, P_i).

This work presents two approaches to choose migrating tasks. In the first strategy that is referred to as the *Continuous* strategy, tasks are allocated to a given core, say p, as non-migrating tasks until a task that cannot fit as a non-migrating task is found. This task is then chosen for splitting. Its first part is allocated to the core p as a C=D task and its second part is allocated to an empty core, say $p + 1$, as a regular task with a phase shift equal to the WCET of the first part of the task. Next, more tasks are allocated to core $p + 1$ until it is full, at which time the next task is split between cores $p + 1$ and $p + 2$, etc.

In the second strategy that is referred to as the *Preselect* strategy, tasks are pre-selected as candidates for migration and remaining tasks are partitioned onto cores. After partitioning, tasks pre-selected for migration are split and allocated, choosing them in non-decreasing order of their relative deadlines. In this strategy, the number of sub-tasks for a given task is not limited to 2. Instead, a task is split as many times as required to make it schedulable.

4.2.1 Calculating C for C=D sub-task

Let us assume that allocation of a task T_s makes core p unschedulable, hence requiring that T_s be split into two sub-tasks, T_s^1 and T_s^2. The process for calculating the WCET of the first sub-task, namely C_s^1, is given below.

1. C_s^1 is initialized to a value such that the total utilization on core p is 1 and D_s^1 is set equal to C_s^1.
2. The value of the test interval, L, is calculated using either Equation 3 or 4 as appropriate.
3. Using QPA Zhang and Burns (2008) as explained in Section 4.1, schedulability analysis is performed backwards from L.
4. If the task set on core p is schedulable, then return (successful allocation); otherwise, a reduced value for C_s^1 is calculated. If this value is 0, then no portion of task T_s can be scheduled on core p.

5. Change the value of L backwards towards time D^p_{min}, the shortest deadline among tasks on core p and repeat QPA (goto step 3).

Let us assume the task set becomes unschedulable for a given value of C^1_s, say at time t. In the interval of time $0 - t$, the demand by tasks that are partitioned (i.e., excluding the C=D task) on a given core p is given by Equation 5.

$$Oth(t) = \sum_{\forall i \in p} \lfloor (t + P_i - D_i)/P_i \rfloor C_i \tag{5}$$

The number of jobs released for the C=D task T_s in the remaining time is given by Equation 6.

$$Num\_jobs = \lfloor (t + P_s - D^1_s)/P_s \rfloor \tag{6}$$

The maximum computation time for each release must be no more than that given by Equation 7.

$$C^1_s = (t - Oth(t))/(\lfloor (t + P_s - D^1_s)/P_s \rfloor) \tag{7}$$

Since D^1_s is set equal to C^1_s, Equation 7 ends up with C^1_s on both sides, thus requiring the iterative calculation shown in Equation 8.

$$C^1_s(r+1) = (t - Oth(t))/(\lfloor (t + P_s - C^1_s(r))/P_s \rfloor) \tag{8}$$

Note that the starting value for C^1_s, namely $C^1_s(1)$, is the one computed such that total core utilization becomes 1.

4.3 EDF-WM Task Splitting Scheme

We now provide a brief description of the semi-partitioned scheduling algorithm proposed by Kato *et al.*, called EDF with Window constraint Migration Kato et al (2009). In this scheme, all tasks on each core are scheduled using EDF and tasks that are allocated to a group of cores (migrating tasks) execute in pre-defined windows of time on each core. The fundamental idea of the algorithm is as follows.

1. Tasks are partitioned onto cores, choosing them in non-increasing order of their densities until no more tasks can be partitioned onto any cores.
2. Among the remaining tasks, one task is chosen at a time (in the same non-increasing order of their densities). The relative deadline of the chosen task is divided equally among the set of cores onto which the task is allocated. In other words, if T_i needs to be split onto n_i number of cores, its relative deadline D_i is divided such that the size of the window for this task on each of the n_i cores is D_i/n_i. Each sub-task is only allowed to execute within its window on its assigned core, thus maintaining sub-task precedence constraints.
3. If all tasks in the task set have been scheduled or if one of them cannot be scheduled even as a split task or migrating task, the algorithm terminates. In the latter case, the task set is deemed unschedulable on the given set of cores.

4.3.1 Splitting of tasks into windows

Let us assume that a task $T_i(\phi_i, C_i, D_i, P_i)$ has to be split into sub-tasks executing within given time windows on a set cores. Initially, the number of cores and, hence, windows, for a task is set to 2. So the sub-tasks are $T_i^1(\phi_i, C_i^1, D_i/2, P_i)$ and $T_i^2((\phi_i + D_i/2), C_i^2, D_i/2, P_i)$. It should be noted that the phase for the second sub-task T_i^2 is shifted in order to maintain precedence constraints between the sub-tasks.

Since the deadline for each sub-task is fixed, the WCET for each sub-task is calculated using processor demand analysis. Let τ^p denote the set of tasks that are partitioned on core p. For a given time interval L, the demand of the partitioned tasks on core p, namely $d(\tau^p, L)$, is calculated using Equation 2, described in Section 4.1.

The number of job releases $Num\_Jobs_i^1$ for the sub-task T_i^1 within the time length L is calculated using Equation 9.

$$Num\_Jobs_i = \lfloor (L + P_i - D_i/2)/P_i \rfloor \tag{9}$$

For the system to be schedulable the, total demand on the system in an interval L, including demand by the originally partitioned tasks and the newly added sub-task, should be less than L, as shown in Equation 10.

$$d(\tau^p, L) + d(\{T_i^1\}, L) \leq L \tag{10}$$

Using PDA for schedulability analysis, the WCET of T_i^1 can be calculated using Equation 11.

$$C_i^1 = (L - d(\tau^p, L))/Num\_Jobs_i \tag{11}$$

Similarly C_i^2 is also calculated. If the sum of C_i^1 and C_i^2 is greater than C_i of task T_i, the splitting of the task is considered successful. Otherwise, the number of cores on which the task is to be allocated is increased by 1 and the same process is repeated after dividing the deadline of the task by the new number of cores.

5 Methodology

In this section, we describe our semi-partitioned scheduling algorithm. The algorithm consists of two main steps, as shown in Listing 1. In the first step, we partition as many tasks as possible onto cores. Tasks that get partitioned are known as *non-migrating* tasks. Any remaining tasks are classified as *migrating* tasks. In the second step, each migrating task is allocated to a set of cores. A migrating task executes on each core it is allocated to in a prescribed order and for a prescribed amount of time. If all tasks are successfully scheduled either as non-migrating or migrating tasks, the task set is deemed schedulable. Otherwise, the task set is declared to be unschedulable.

We first present our scheme for allocating migrating tasks and then describe two partitioning strategies that are specifically suited to our architectural model and our approach for allocating migrating tasks, respectively.

5.1 Allocation of Migrating Tasks : Slack Based Semi-partitioning (SBS)

The subset of tasks that cannot be partitioned using a given partitioning strategy are considered as candidates for migration. We first present necessary terms and theorems and then present the algorithm for the allocation of migrating tasks onto a set of cores.

Definition 1 Slack time Δ_i^c for a migrating task T_i on a core c is defined as shown in Equation 12.

$$\Delta_i^c = \max(U_{max}^c * D_{min}^c - \sum_{j \in p(c)} C_j^c, 0) / \max(\lfloor \frac{P_{min}^c}{P_i} \rfloor, 1) \tag{12}$$

Here, $p(c)$ is the set of non-migrating tasks allocated on core c and C_j^c is the WCET of task T_j on core c, based on locked regions it retains on core c. D_{min}^c is the shortest relative deadline among non-migrating tasks allocated to core c, given by $\min_{j \in p(c)} D_j$, and P_{min}^c is the period of this task with shortest relative deadline. U_{max}^c is the utilization cap for a core. We incorporate this aspect because it is sometimes useful, from a power/energy consumption standpoint, to run cores below 100% load. On a core with maximum available utilization of 1 where the period of the migrating task is greater than or equal to the period of the task with shortest relative deadline, Δ_i^c is simply $\max(D_{min}^c - \sum_{j \in p(c)} C_j^c, 0)$.

The intuition behind this theorem is that consides the worst case when all tasks release in phase. If all tasks partitioned on a core executes within the time interval which is less than the minimum period of the tasks then there would be some free processor time which would be left within every such smallest period interval. The minimum of those free processor times on a core is termed as the slack time of the core.

Theorem 1 *If a migrating task T_{mig} has a slack time of Δ_{mig}^c on core c, it is guaranteed to get the highest priority for Δ_{mig}^c amount of time on core c within any time interval equal to the relative deadline of the task with the shortest relative deadline among non-migrating tasks allocated to core c and is guaranteed not to violate schedulability of non-migrating tasks on core c.*

Proof.

Part 1. Suppose a task T_{mig} arrives at time t_{mig}. T_{mig} has relative deadline of Δ_{mig}^c on core c. Let us assume that T_{mig} does not get highest priority when it arrives. Let $d_{non-mig}$ and $e_{non-mig}^{left}$ be the absolute deadline and the remaining execution time of the current highest priority non-migrating task $T_{non-mig}$ on the core. This implies that

$$d_{non-mig} < t_{mig} + \Delta_{mig}^c \tag{13}$$

$$e_{non-mig}^{left} > 0 \tag{14}$$

From Equation 12, we know that in a period P_{min}, we always have slack time of Δ_{mig}^c. So the effective slack time $\Delta_{non-mig}$ of task $T_{non-mig}$ will be at least equal to $\lfloor P_{non-mig}/P_{min} \rfloor * \Delta_{mig}^c$. Since $P_{non-mig} \geq P_{min}$, $\Delta_{non-mig} \geq \Delta_{mig}^c$.

Let us assume that T_{mig} gets highest priority after the execution of $T_{non-mig}$. So, the slack time $\Delta_{non-mig}$ left after the execution of $T_{non-mig}$ is $d_{non-mig} - (t_{mig} + e^{left}_{non-mig})$. From Equation 13, we get $\Delta_{non-mig} < \Delta^c_{mig} - e^{left}_{non-mig}$ From Equation 14, $\Delta_{non-mig} < \Delta^c_{mig}$. This is a contradiction since $\Delta_{non-mig} \geq \Delta^c_{mig}$.

Part 2. Since non-migrating tasks are allocated to a core only if they are schedulable according to the EDF schedulability test, any deadline miss must be due to the arrival and immediate execution of T_{mig}. By definition, even the task with the shortest relative deadline can accommodate a slack of Δ^c_{mig} under the worst-case scenario that it is delayed by one job of every other task in the set of non-migrating tasks. Thus, no task can miss its deadline due to the execution of T_{mig}.

Algorithm. We now describe the various steps of our algorithm for allocating migrating tasks onto a set of cores (Lines 2 - 2 of Listing 2).

Listing 1 Task Allocation Algorithm

```
1:  Task_Allocator(Task_List, Cores)
2:  Partition_Tasks(Task_List, Cores)
3:  if (Tasks not Empty) then
4:      schedulable ← Allocate_Migrating_Tasks(Task_List, Cores)
5:  end if
6:  if (schedulable) then
7:      RETURN (SUCCESS)
8:  else
9:      RETURN (FAILURE)
10: end if
```

Step 1. Calculate Slack Times. The slack time available for a given migrating task on each core is calculated according to Equation 12.

Step 2. Allocate Task Portion. Cores are sorted in *non-increasing* order of the slack time available on them and a portion of the migrating task is allocated to the core with the maximum slack time. That core is then removed from further consideration by the algorithm since we allow only one migrating task to be allocated to a given core. These steps are shown in Lines 2 - 2 of Listing 2. The relative deadline of this portion of the migrating task is set to be equal to the available slack time on the core. In accordance with Theorem 1, the migrating task is guaranteed to execute at the highest priority on this core without violating schedulability of non-migrating tasks on the core . So, the portion of the migrating task effectively executes at 100% utilization for a duration equal to the available slack time. This method has the advantage that the underlying scheduling policy (EDF) and, hence, the corresponding schedulability test, remain unchanged.

Step 3. Calculate Remaining Task Utilization. The remaining WCET of the migrating task, including the overhead of migration is calculated using Equation 15 (Lines 2 - 2 of Listing 2).

$$C_i^{rem,m} = C_i^{rem,m-1} - \max_{c \in cores} \Delta_i^c + M_i^m \tag{15}$$

Here, $C_i^{rem,m}$ is the WCET remaining after the m^{th} migration, $cores$ is the set of cores currently available for consideration and M_i^m is the migration overhead for the

Listing 2 Allocation of Migrating Tasks

```
1: Allocate_Migrating_Tasks(Migrate_List, Cores)
2: while (Migrate_List not Empty) do
3:    Task ← Choose_Task_by_Ordering_Scheme(Migrate_List)
4:    Done ← FALSE
5:    while (Done ≠ TRUE) do
6:       Core ← Find_Max_Slack_Core(Cores)
7:       if (Valid(Core)) then
8:          Allocate_Task_To_Core(Task, Core)
9:          Delete_Core(Cores, Core)
10:      else
11:         RETURN FALSE
12:      end if
13:      Mig_Curr ← Calc_Curr_Migration_Overhead(Task)
14:      u_Task ← Calc_Rem_Util(Task, Mig_Curr)
15:      Mig_Last ← Calc_Last_Migration_Overhead(Task)
16:      if (u_Task + Mig_Last ≤
         (1 - Max_Util_Core(Cores))) then
17:         Core ← Min_Util_Min_Slack_Core(Cores, Task)
18:         if (Valid(Core)) then
19:            Allocate_Task_To_Core(Task, Core)
20:            Delete_Task(Migrate_List, Task)
21:            Delete_Core(Cores, Core)
22:            Done ← TRUE
23:         else
24:            RETURN FALSE
25:         end if
26:      end if
27:   end while
28: end while
29: RETURN TRUE
```

m^{th} migration, calculated using Equation 16.

$$M_i^m = (Read + Write + \sum_{h=1}^{nh(src^m, dst^m)} l_h) * nl_i^{migrated} \tag{16}$$

Here, src^m and dst^m are the source and destination cores for the m^{th} migration. $Read$ and $Write$ are the latencies for reading from and writing to a cache line, respectively, at the source and target of the migration. $nh(src^m, dst^m)$ is the number of hops between the source and destination of the migration, l_h is the latency of migration of a cache line over a single hop and $nl_i^{migrated}$ is the number of lines migrated (subset of nl_i^{locked}). The remaining utilization of the migrating task is calculated using the remaining WCET and the time available before the migrating task's deadline. If this remaining utilization plus the overhead of migrating the task back to the core where its first portion is allocated (for the next job) can be accommodated on some core (Lines 2 - 2 of Listing 2), this remaining portion becomes the last portion of the task. Otherwise, steps 2 and 3 are repeated until the remaining utilization of the migrating task under consideration is less than or equal to the available utilization on some core.

Due to the use of the TDM approach described in Section 2, physical core locations do not affect the WCETs of tasks. Hence, our algorithm uses virtual core numbers and we assume that migrating tasks are allocated onto physically neighboring cores, hence making the number of hops small (equal to 1 when possible). Since we allow only one migrating task per core, mapping of virtual to physical cores effectively is straightforward.

Step 4. Choose Core for Last Task Portion. If more than one core can accommodate the last portion of a migrating task, we choose the one with the minimum current utilization. If more than one core has the same current utilization, we choose the one with the minimum slack time. These steps are shown in Lines 2 - 2 of Listing 2. Both these non-greedy heuristics are in an effort to ensure that cores with larger utilizations and larger slack times are available for other migrating tasks.

Steps 1, 2, 3 and 4 are repeated for each migrating task until either all of them are successfully allocated or no more cores remain for consideration. In the latter case, the task set is declared unschedulable on the given number of cores. Note that, since we dedicate one cache way on each core for migrating tasks, it always retains its chosen locked regions on all cores to which it may be allocated.

Algorithm Complexity. For each migrating task, our algorithm iterates over the set of all cores in the worst case. So, the complexity is O(k*m), where k is the number of migrating tasks and m is the number of cores.

Comparison with Existing Work. In our work, we specifically target independent cache based systems with support for cache locking. In contrast, some existing techniques (Dorin et al, 2010; Kato and Yamasaki, 2008) assume a cacheless system and some others (Kato et al, 2009; Burns et al, 2010) assume a shared cache based system. Hence, they do not consider the overheads of migrating cache content. This results in a significantly higher baseline for existing techniques in terms of system utilization and schedulability.

An advantage of our algorithm for allocating migrating tasks compared to a semi-partitioned approach proposed in related work (Dorin et al, 2010) is that we maintain the periodicity of tasks in contrast to that work where each job of a migrating task over the entire hyperperiod must be explicitly considered. Other related work (Kato and Yamasaki, 2008) uses a restricted migration model, thus avoiding the overhead of migrations within a single job of a task. However, the method requires a modification to the underlying scheduling policy and, hence, the schedulability test.

Two recent semi-partitioning schemes proposed by Kato *et al.* (Kato et al, 2009) and Burns *et al.* (Burns et al, 2010) employ a task-splitting approach to semi-partitioned scheduling like we do in our current paper. Both these existing schemes could potentially identify and utilize more slack time on a core than our proposed scheme for scheduling migrating tasks. However, neither of these schemes explicitly considers migration overheads. Furthermore, both these schemes use schedulability tests based on demand bound analysis. In the worst case, such a schedulability test could require exact calculations for the entire hyperperiod of a task set and, hence, is computationally much more complex.**We demonstrate the difference in the semi-partitioned approaches proposed by Kato *et al.* (Kato et al, 2009), Burns *et al.* (Burns et al, 2010) and SBS using a running example next.**

5.2 Running Example Analysing Semi-Partitioned Scheme

In this section we demonstrate the difference between the three approaches of semi-partitioning mentioned before in this paper. We describe the differences using a dual core system with a common partitioned set of tasks. Table 1 describes the task set charecterisitics used in this example.

Table 1 Running Example Demonstrating Semi-Partitioned Approaches

i	**Period**(P_i)	**Deadline**(D_i)	**WCET**(C_i)
1	100	100	40
2	100	100	40
3	200	200	40
4	200	200	20
5	100	100	60

Table 2 describes the state of the system after partitioning of tasks. The first coulumn represents the core IDs, second coulumn represents the set of partitioned tasks on each core and the last coulumn represents the current utilization of the cores. Please note that task T_5 cannot be partitioned on any of the two cores. Next we describe the semi-partitioning of task T_5 using the three approaches mentioned before.

Table 2 State of System After Partitioning

Core	**Partitioned Tasks**	**Utilization**
1	T_1,T_3	0.6
2	T_2,T_4	0.5

5.2.1 C=D Task Splitting Scheme

In the running example the value of L calculated both for core 1 and 2 are 200. So, $Oth(200)$ is calculated using Equation 5 and is equal to 120 from Equation 17.

$$Oth(200) = \lfloor(200 + 100 - 100)/100\rfloor 40 + \lfloor(200 + 200 - 200)/200\rfloor 40 \quad (17)$$

The first subtask C_5^1 is calculated as 60 using Equation 7 for t as 200 in Equation 18.

$$C_5^1 = (200 - Oth(200))/(\lfloor(200 + 100 - C_5^1)/100\rfloor) \quad (18)$$

So using C=D scheme, the task T_5 will be allocated to core 1 and execute as task T_5^1 having C_5^1 as 60 and D_5^1 as 60.

Table 3 Example Task Set

i	P_i	C_i^{locked}	u_i^{locked}	nl_i^{locked}
1	10000	7000	0. 7	250
2	10000	6000	0. 6	200
3	50000	30000	0. 6	150
4	40000	24000	0. 6	200
5	50000	30000	0. 6	250
6	50000	30000	0. 6	150
7	50000	30000	0. 6	100
8	100000	60000	0. 6	100
9	100000	60000	0. 6	150
10	100000	60000	0. 6	150

5.2.2 EDF-WM Task Splitting Scheme

In this scheme we divide the task to be semi-partitoned T_5 into two windows each having relative deadlines of 50.The demand of partitioned tasks denoted as $d(\tau^p, L)$ on core 1 is calculated using Equation 10. Using $d(\tau^p, L)$ we calculate the WCET C_5^1 of subtask T_5^1 using Equation 11. We calculate C_5^1 for three values of L (200,100 and 50). This is shown in Equations 19,20,21.

$$C_5^1 = (200 - d(\tau^p, 200))/Num\_Jobs_5 \tag{19}$$

$$C_5^1 = (100 - d(\tau^p, 100))/Num\_Jobs_5 \tag{20}$$

$$C_5^1 = (50 - d(\tau^p, 50))/Num\_Jobs_5 \tag{21}$$

The values of C_5^1 calculated from the above Equations 19,20 and 21 are 60,60 and 50 respectively. The final value of C_5^1 is the minimum of all the possible values and hence equal to 50.

Thus using this method task T_5 is split into two subtasks $T_5^1(50, 50, 100)$ allocated to core 1 and $T_5^2(10, 50, 100)$ allocated to core 2.

5.2.3 SBS Task Splitting Scheme

In this scheme the slack calculated on core 1 is 20 using Equation 12. The subtask of task T_5 allocated on core 1 is $T_5^1(20, 20, 100)$. The utilization of the remaining subtask $T_5^2(40, 80, 100)$ is equal to the available utilization on core 2. So subtask T_5^2 is allocated on core 2.

We observe that SBS scheme calculates the minimum slack among all of the three approaches because we use a non-optimal but mathematically much simplified schedulability test(EDF schedulability test). In order to improve our slack time calculation on any core we have designed partitoning approaches which will be described in the next section.

It should be noted that we did not include the migration overhead calculation in our running example as it would complicate the calculations and hence hinder the understanding of the difference in the basic approach of semi-partitioning among the three algorithms compared in this paper.

Listing 3 Architecture Aware Partitioning(AAP)

```
1: Partition_Tasks(Task_List, Cores)
2: while (Task_List not Empty) do
3:    Task ← Choose_Task_by_Ordering_Scheme(Tasks)
4:    Core ← Min_Util_Change_Core(Task, Cores)
5:    if (Util(Core) + Util(Task) ≤ 1) then
6:       Allocate_Task_To_Core(Task, Core)
7:       Delete_Task(Tasks, Task)
8:    else
9:       Add_Task(Migrate_List,Task)
10:      Delete_Task(Task_List, Task)
11:   end if
12: end while
```

5.3 Task Partitioning

There are two main decisions involved in any partitioning strategy, namely the order in which tasks must be chosen for allocation (*task ordering*) and the core to which a chosen task must be allocated (*task allocation*). We now present two task ordering schemes and two task allocation schemes.

5.3.1 Task Ordering

Utilization based ordering. In this scheme, tasks are chosen in non-increasing order of their utilizations. The purpose of this scheme is to let as many tasks with higher utilization get partitioned and leave the tasks with comparatively lower utilization as candidates for migration.

Migration characteristics based ordering. In this scheme, we use a heuristic based on the migration characteristics of a task since lower utilization tasks may not always be better candidates for migration. We use the term migration-slack ratio to capture the migration characteristics of a task. Let us assume that the overhead of locked cache migration for task T_i over one hop is M_i. Then, the migration-slack ratio, $T_i^{mig-slack}$ is calculated using Equation 22.

$$T_i^{mig-slack} = M_i/(D_i - C_i) \tag{22}$$

The migration-slack ratio of a task captures the number of one-hop migrations the task can endure within its relative deadline. The larger the ratio, the smaller the number of migrations a task can endure. Hence, we order tasks in non-increasing order of their migration-slack ratios in an effort to let as many tasks with less tolerance to migrations get partitioned, leaving those with higher tolerance as candidates for migration.

5.3.2 Task Allocation

We first present an algorithm that partitions tasks in a cache- and NoC-aware manner. Next, we present a look-ahead scheme that partitions tasks with the goal of improving overall system schedulability in the context of our scheme for allocating and scheduling migrating tasks.

5.3.3 Architecture Aware Partitioning (AAP)

Our Architecture Aware Partitioning algorithm (AAP), shown in Listing 3, sorts tasks using one of the task ordering schemes presented in Section 5.3.1. It then chooses suitable cores for them one at a time, using a *worst-fit* strategy. If the worst-fit strategy results in several candidate cores for a task, the task is allocated to 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. The reasons for these choices are explained below.

When a new 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 regions to resolve the conflicts, thereby increasing the utilizations of these tasks. The change in WCET of a task resulting from a reduction in its locked cache lines is calculated as the product of the number of accesses to each line being unlocked and the time taken to fetch the line. The (location independent) memory access latency for each line is calculated using the NoC routing and arbitration model presented in Section 2. In order to minimize the change in task utilizations due to cache conflicts, our algorithm chooses to unlock conflicting regions with the lowest number of accesses (also called minimum access frequency).

In such a situation, using a worst-fit strategy provides the highest likelihood of actually being able to schedule the task on an initially chosen core since it allows for more WCET inflation due to potential cache conflicts. Among feasible cores, a task is allocated to the core that suffers the minimum change in utilization due to the allocation. This is done in order to minimize the additional off-chip memory traffic generated due to cache region unlocking which, in turn, could lead to savings in power/energy consumption. If two or more cores have the same change in utilization, the core with the minimum utilization among them is chosen. This is done to improve load balancing and, hence, minimize thermal degradation of cores. If the new task cannot be accommodated on any core due to utilization bounds, it becomes a candidate for migration.

Illustrative Example.

We explain our algorithm AAP integrated with the slack-allocation algorithm for migrating tasks by the help of a simple example task set whose characteristics are shown in Table 3. The first column shows the task ID and the second column shows the period of the task. **In this experiment the period of a task is equal to its relative deadline.** The third and fourth columns show the locked WCET (when all of a task's chosen lines are locked) and the corresponding utilization, respectively. The last column shows the number of locked lines for the task. We assume a 9-core (3 X 3) 2D mesh in this example. In this example, we use the utilization based task ordering scheme.

Step 1. In the example shown in Table 3, since there are 10 tasks to be allocated onto 9 cores, tasks 1 to 9 get allocated onto the 9 cores according to our algorithm and each task is allowed to retain its chosen locked regions due to the absence of conflicts. When we try to schedule task 10 that has a (locked) utilization of 0. 6, we find that

Table 4 Core Slack Times After Partitioning

c	i	P_i	C_i^c	Δ_i^c
1	1	10000	7000	3000
2	2	10000	6000	4000
3	3	50000	30000	20000
4	4	40000	24000	16000
5	5	50000	30000	20000
6	6	50000	30000	20000
7	7	50000	30000	20000
8	8	100000	60000	40000
9	9	100000	60000	40000

it cannot be accommodated on any of the cores. Hence, this task is considered for execution as a migrating task.

Step 2. Table 4 shows the slack times of cores for our running example. The first column shows the core ID and the second column shows the IDs of non-migrating tasks allocated to the core. The third and fourth columns show the periods and WCETs of tasks allocated to the core, respectively (repeated for convenience) and the last column shows the slack time of the core.

Step 3: A portion of the migrating task 10 is allocated to core 8 since core 8 has the maximum slack time (the tie between cores 8 and 9 that have the same slack time is resolved using the core ID).

Step 4: Task 10 is allocated on core 8, on which it gets a continuous execution interval of 40000 units since, according to Theorem 1, it executes at the highest priority for that duration. Its remaining execution time is 20000 units and the remaining time before its deadline is 60000 units. As seen from Table 3, the number of locked lines for task 10 is 150. The overhead for migrating one cache line is assumed to be 10 cycles. This causes an overhead of 1500 cycles, which is added to the remaining WCET, according to Equation 15. We then check if this updated utilization plus another migration overhead to account for the task's return to core 8 can be accommodated on some core. If so, this portion becomes the last portion. In our example, a further overhead of 1500 cycles is added and the updated remaining utilization is (20000 + 1500 + 1500)/(60000) = 0. 38, which can be accommodated on several cores.

Step 5: We find that the last portion of task 10 (with utilization 0. 38), can be accommodated on cores 1, 2, 3, 4, 5, 6, 7 or 9. Core 8 is eliminated from consideration since the first portion of task 10 has already been allocated to it. In accordance with the non-greedy heuristics used in our algorithm, core 2 is chosen to host the last portion of task 10.

Algorithm Complexity. The partitioning stage of our algorithm iterates over tasks, and, for each task, iterates over cores to check for possible allocations. So the complexity for this partitioning approach is $O(n * m)$ where n is the total number of tasks and m is the total number of cores. If we combine the complexity of both AAP and that for the allocation of migrating tasks we get an overall complexity of $O(n * m)$, which is acceptable since this is an offline algorithm.

Applicability of AAP. AAP is particularly suited to task sets in which several tasks have high utilizations (≥ 0.5) and a few tasks have lower utilizations. Due to

Table 5 Synthetic Task set

i	P_i	C_i^c	u_i
1	1000	500	0.5
2	800	320	0.4
3	100	40	0.4
4	100	40	0.4
5	90	27	0.3

Table 6 Allocation by slack-agnostic heuristics

c	i	P_i	C_i^c	Δ_i^c	u^{left}
1	1	1000	500	500	0.5
2	2	800	320	480	0.6
1	3	100	40	0	0.1
2	4	100	40	0	0

high utilizations (possibly increased due to conflicts in locked cache regions), partitioning may be able to accommodate only one or two tasks on each core. Although the remaining tasks have high enough utilizations that they cannot be partitioned onto cores, if some of them have shorter WCETs, it is likely that there are cores on which there is sufficient slack time to accommodate such tasks.

Strengths of AAP. The heuristics used in AAP help maximize the gains of cache locking on cores and improve load balancing.

Limitations of AAP. Since AAP mainly focuses on maximizing cache locking for a given task set, it may lead to decrease the total slack time available in the system in order to achieve close to optimal cache locking. This would decrease the schedulability of semi-partitioned tasks on the system. And hence the total schedulable utilization is affected.

5.3.4 Motivation for Slack Awareness in Partitioning

In this section, we demonstrate the usefulness of a slack-aware partitioning scheme in overcoming the limitations of AAP. Consider the synthetic task set in Table 5 and assume that this task set must be allocated on a dual core system.

First, we partition tasks using AAP. Let us assume that tasks are arranged in non-increasing order of their (locked) utilization, resulting in the order 1, 2, 3, 4 and 5. Tasks are then allocated to cores using the minimum utilization change and minimum utilization heuristics. In accordance with these heuristics, tasks 1 and 2 are allocated to cores 1 and 2, respectively, task 3 is allocated to core 1 and task 4, to core 2. The final allocation is shown in Table 6. The slack time column (Δ_i^c) in Table 6 represents the slack time generated after the given task allocation on the core **and the last coulumn (u^{left}) represents the available utilization after the allocation of a specific task.** We observe that task 5 is not partitioned using these heuristics, thus leaving it as a candidate for migration. We also observe that the slack time, calculated as described in Section 5.1, is 0 for both cores. As a result, task 5 is unschedulable.

Next, we partition tasks considering the amount of slack time the partition would result in on various cores. A simple way to achieve this is to allocate tasks with

Table 7 Allocation by slack-aware heuristics

c	i	P_i	C_i^c	Δ_i^c	u^{left}
1	1	1000	500	500	0.5
1	2	800	320	0	0.1
2	3	100	40	60	0.6
2	4	100	40	20	0.2

similar relative deadlines to the same core. One such allocation, for the task set shown in Table 5, is shown in Table 7. Since task deadlines are taken into account during partitioning, tasks 1 and 2 get allocated to core 1 and tasks 3 and 4, to core 2. We now find that there is a slack time of 20 units on core 2. Part of task 5 is scheduled in this slack time and the remaining utilization of task 5 is 0.1, which can be scheduled on core 1 as a normal task.

From the above example, we see that a slack-aware partitioning of tasks can lead to increased slack time and increased overall schedulability, thus motivating the algorithm we present next.

5.3.5 Slack and Architecture Aware Partitioning (SAAP)

The primary goal of the partitioning strategy presented in this section is to allocate tasks in a way that improves the chances of successfully scheduling migrating tasks using the scheme presented in Section 5.1, and to thereby improve overall system schedulability.

The success of scheduling a given migrating task to a given set of cores using with the scheme presented in Section 5.1 depends on two factors, namely the migration characteristics of the task and the slack time available on the cores. We propose a partitioning scheme that allocates tasks in a way that maximizes slack time on cores as much as possible. Listing 4 shows our proposed algorithm.

Listing 4 Slack- and Architecture Aware Partitioning(SAAP)

```
1: Initialize_System(Core_List,Task_List)
2: Label all Cores as 0
3: Arrange_Tasks_with_Highest_Relative_Deadline_First(Task_List)
4: System_Median ⇐ Pick_Middle_Task(Task_List)
5: while (Task_List ≠ Empty) do
6:    Task ← Choose_Task_by_Ordering_Scheme(Task_List)
7:    if Task_Deadline ≥ System_Median then
8:       Allocate_Tasks_above_System_median(Task, Cores)
9:    else
10:      Allocate_Tasks_below_System_median(Task, Cores)
11:   end if
12:   Delete_Task(Task_List, Task)
13: end while
```

Task Allocation According to Theorem 12, the slack time on a core is calculated by subtracting the sum of WCETs of all tasks on the core from the minimum relative

Listing 5 Allocation of Tasks Above System Median

```
1: Allocate_Tasks_above_System_median(Task, Cores)
2: Core_List ← Find_List_of_Schedulable_Cores(Task,Cores_Labelled_2)
3: if Core_List ≠ empty then
4:    min_slack_decrease_cores ← Find_Cores_with_minimum_slack_decrease_ratio(Core_List)
5:    Core ← Min_Util_Change_Core(Task, min_slack_decrease_cores)
6:    Allocate_Task_To_Core(Task, Core)
7: else
8:    if Cores_Labelled_0 ≠ empty then
9:       Allocate_Task_to_empty_core(Task,Core)
10:      Label_empty_core_2(Core)
11:   else
12:      Core_List ← Find_List_of_Schedulable_Cores(Task,Cores_Labelled_1)
13:      if (Core_List ≠ empty) then
14:         Core = Find_Core_with_minimum_slack_decrease_ratio(Core_List)
15:         Allocate_Task_To_Core(Task, Core)
16:      else
17:         Add_Task(Migrate_List,Task)
18:      end if
19:   end if
20: end if
```

Listing 6 Allocation of Tasks Below System Median

```
1: Allocate_Tasks_below_System_median(Task, Cores)
2: Task ← Choose_Task_by_Ordering_Scheme(Task_List_below_System_median)
3: Core_List ← Find_List_of_Schedulable_Cores(Task,Cores_Labelled_1)
4: if Core_List ≠ empty then
5:    min_deadline_decrease_cores ← Find_Cores_with_minimum_deadline_decrease(Core_List)
6:    Core ← Min_Util_Change_Core(Task, min_deadline_decrease_cores)
7:    Allocate_Task_To_Core(Task, Core)
8: else
9:    if Cores_Labelled_0 ≠ empty then
10:      Allocate_Task_to_empty_core(Task,Core)
11:      Label_empty_core_1(Core)
12:   else
13:      Core_List ← Find_List_of_Schedulable_Cores(Task,Cores_Labelled_2)
14:      if (Core_List ≠ empty) then
15:         Core = Find_Core_with_minimum_slack(Core_List)
16:         Allocate_Task_To_Core(Task, Core)
17:      else
18:         Add_Task(Migrate_List,Task)
19:      end if
20:   end if
21: end if
```

deadline among those tasks. Since tasks with shorter relative deadlines provide low chances of reasonable slack times, we divide tasks into two groups, namely those with shorter relative deadlines and those with longer relative deadlines, and allocate these groups onto different sets of cores as far as possible, attempting to maximize slack time on cores containing tasks from the second group.

Division of Tasks into Groups. We employ a straightforward approach to divide tasks into two groups. We calculate the mathematical median of task relative deadlines by arranging them in non-decreasing order and choosing the deadline value that

lies in the middle of the sorted list. If there are two values in the middle, we use their average. Henceforth, we refer to this value as the *system median*. Tasks with relative deadlines below the system median are part of the first group and those above form the second group.

Cores are categorized based on the tasks allocated to them. Cores that contain only tasks from the second group (tasks with deadlines higher than the system median) are labeled 2 and cores that contain at least one task from the first group are labeled 1. Empty cores are labeled 0. All cores initially start with a label of 0.

Allocation of Tasks Above System Median. Listing 5 shows the steps used to allocate a task above the system median to a core. The steps performed for a given task T_i are described below. We first identify cores on which T_i is schedulable among cores with label 2 (cores with only tasks with relative deadlines higher than the system median). If no such core is available, we choose a core with label 0 (empty core) if one is available, allocate T_i to that core and re-label that core with a label 2. On the other hand, if there are suitable cores with label 2, we use the heuristics listed below to choose one among these cores, in the specified order.

Heuristic A1. We choose the core where the allocation of the task under consideration leads to the minimum decrease in the core's slack time. The effect of a task allocation on the slack time of a given core is estimated using Equation 23.

$$SD_i^c = (D_c^{min} - C_i^c)/\max(1, D_c^{min} - D_i) \tag{23}$$

Here, D_c^{min} is the shortest relative deadline on core c before the allocation of T_i and C_i^c is the WCET of T_i on core c. Note that the WCET of a task depends on the core to which it is allocated due to cache conflicts among tasks allocated to a given core[3]. A higher value of SD_i^c indicates a smaller decrease in the slack time of the core under consideration. Hence, a task is allocated to a core where it has the highest value of SD_i^c.

Heuristic A2. If there is more than one core where task T_i has the same value of the ratio calculated using Equation 23, we choose the core that leads to the minimum change in utilization (due to unlocking of cache lines due to potential conflicts) for T_i.

Heuristic A3. If there are multiple cores that result in the same (minimal) change in utilization for T_i, it is allocated to the core with the lowest current utilization. This is done in an effort to improve load balancing among cores.

If no suitable core with label 2 or a core with label 0 is available for T_i, it is allocated to a core with label 1 using Heuristics A1, A2 and A3 described above.

Allocation of Tasks Below System Median. Listing 6 shows the steps used to allocate a task with relative deadline below the system median to a core. The steps performed for a given task T_i are described below.

We first identify cores on which T_i is schedulable among cores with label 1 (cores with at least one task with relative deadlines lower than the system median). If there are no such cores, we choose a core with label 0 (empty core), allocate T_i to the empty core and re-label to core to 1. On the other hand, if there are multiple suitable cores

[3] Note that the WCET of a task is not dependent on the physical location of a core due to the use of our weighted TDM approach, but just on the tasks allocated to the core.

with label 1, we choose one using the heuristics described below, in the specified order.

Heuristic B1. We choose the core where the allocation of task T_i leads to the minimum change in the smallest relative deadline on that core. This is in an effort to maximize the potential slack time on cores.

Heuristic B2. If there are multiple cores that suffer the same change in smallest relative deadline, we choose the core where task T_i suffers the smallest change in its utilization.

Heuristic B3. If there are multiple cores where T_i suffers the same (minimal) change in its utilization, we choose the core with the smallest current utilization in an effort to balance the load among cores.

If no suitable core with label 1 or label 0 is available for T_i, we identify cores with label 2 where T_i is schedulable. If there are multiple such cores, we choose the one with the smallest current slack time, allocate T_i to that core and re-label that core to 1.

Once all tasks below the system median have been allocated, if there are still unallocated tasks above the system median, they are allocated to cores labeled 1 if possible. This is done in order to minimize the tasks on the cores labeled 2 and, hence, maximize slack time on these cores.

6 Discussion And Practical Implementation

In this section we discuss the possibility of implementation of the proposed algorithm on any available architecture. The target architecture for the proposed algorithm is Tilera Pro64 like architecture. The Network-on-Chip implemented on Tilera consists of dedicated bidirectional channel for core to core communication. This channel can be used by the migration traffic generated because of the semi-partitioned tasks on cores.Since, the number of migrating tasks are limited to one per core, we place the set of cores allocated to one migrating task such that it does not interfere with the migration traffic of another task.

Also, the off-chip memory access traffic does not interfere with the migration traffic as there is a dedicated bi-directional channel for off-chip memory access. Although Tilera does not support TDMA on the offchip memory traffic, we can arbitrate control of a memory controller between the sharing cores in software.

Cache locking is also not supported in Tilera but we can use its feature of memory pinning to decide what contents of the cache becomes irreplacable. So we can implement our algorithm on Tilera Pro64 after making the features compatible to its implementation.

However this will introduce additional overhead since most of the change is controlled through software. In order to minimize these additional overheads, the ideal platform well suited for the implementation of this algorithm would be Tilera Pro64 like architecture with additional features like cache locking and TDMA on the off-chip memory access link of the NoC, implemented on it.

Table 8 System Configuration

Parameter	Configuration
Processor Model	in-order
Cache Line Size	32Bytes
L1 D-Cache Size/Associativity	256KB/4-way
L1 hit latency	1 cycle
Replacement Policy	Least Recently Used
Number of Cores	9
Cache to cache Transfer latency	13 cycles
External Memory Latency	90 cycles

7 Experimental Setup

The architectural configuration used in our simulations is shown in Table 8. The cache to cache transfer latency is 13 cycles (2 cycles to read the source cache line, 4 cycles for transferring the cache line over one hop and 7 cycles to write to the target cache). The external memory read latency shown in the last row is the sum of **1)** the NoC latency for the request to travel from the core to the memory controller (**6 cycles**, calculated using the NoC routing and arbitration model described in Section 2), **2)** the latency of the actual memory access once the request reaches the memory controller (**60 cycles** - this latency includes the delay due to pipelining of multiple memory pending requests at the memory controller), and **3)** the NoC latency for the returned 32-byte cache line to travel back from the memory controller to the requesting core (**24 cycles** - different from the latency for the memory request to travel to the memory controller due to difference in the size of the transferred data). The external memory write latency is analogous. We assume that all four ways of the L1 data cache are lockable, among which one way is reserved for locking the footprint of the (single) migrating task allocated to a given core. In our current experiments, we have only modeled data cache migration. Note that this is just a choice for our experimental setup. Our methodology itself is not limited to any one kind of cache.

Note that the C=D task splitting scheme and the EDF-WM scheme (Section 3.3.1) do not explicitly consider cache content migration and overheads resulting thereof. In order to enable a fair comparison, we have modified these two schemes to account for migration overheads in the same way as that used by the slack based semi-partitioned scheduling scheme (Section 5.1). Hereafter, we refer to the three schemes as C=D, EDF-WM and SBS, respectively.

For all schemes, the number of migrating tasks on a core is restricted to 1 and one way of the cache of every core is dedicated for the locked lines of the migrating task. These restrictions are imposed in order to ensure that there is no contention among migration traffic resulting between cores and to ensure that a migrating task always has room to lock its cache contents. This enables calculation of tight bounds on task migration overheads.

7.1 Task Set Generator

We use synthetic task sets in order to compare the performance of different algorithms. The synthetic task sets are generated using an unbiased random task set generator for multi-core architectures (R.I.Davis and A.Burns, 2011). This method is an extension of the UUnifast algorithm proposed by Bini et al. (Bini and G.C.Buttazzo, 2005).

The reason we chose to do this is so that we could conduct sensitivity studies on how variations of the number of accesses (termed access frequency) to given memory chunks, task utilizations and cache conflict patterns for tasks in a task set affect the performance of the three algorithms. **We generate task sets that have a total locked utilization of 9 (since the total number of cores in our experimental setup is 9). By locked utilization we mean the utilization of a task with a fully locked WCET. Assuming that all tasks can lock all theor cache footprints in the cache, we can have maximum schedulable utilization as 9. This serves as a baseline to compare the performance of our algorithms in the experiments.** In each task set, tasks in one of the deadline ranges 1000-10000,10000-20000,20000-30000 and 30000-40000 are generated. We generate 50 task sets for each deadline range for every set of experiments. Also the task memory footprints are generated such that they overlap in cache to demonstrate the effects of cache conflicts.

The memory footprint of a given task is divided into chunks where each chunk has a fixed number of elements and a known access frequency. The access frequency of a chunk indicates the total number of times an element in that chunk is accessed, so the access frequency in the task set generator is generated in proportion to the number of elements in the chunk. Since we use locked caches, for a chunk that is not locked in the cache, all accesses are considered misses. Hence, the access frequency to a chunk plays an important role.

8 Experimental Results

In this section, we present results obtained through simulations with task sets constructed using real benchmarks and synthetically generated task sets. We first demonstrate the effectiveness of our semi-partitioned scheduling algorithm over a purely partitioned approach. We then compare the performances of the task partitioning and ordering schemes proposed in this paper, used in conjunction with our scheme for scheduling migrating tasks, against the recent semi-partitioned scheduling algorithms (Burns et al, 2010; Kato et al, 2009) explained in Section 3. All execution time, period and deadline values are reported in terms of the number of processor cycles.

8.1 Comparison with Partitioned scheduling

In this section, our goal is to demonstrate the effectiveness of our semi-partitioned scheduling algorithm over a purely partitioned approach. Here, we use the utilization based task ordering scheme presented in Section 5.3.1 and the AAP strategy presented in Section 5.3.3 for task allocation.

We use benchmarks from the DSPStone benchmark suite (Zivojnovic et al, 1994) to construct task sets. Table 9 shows the details of tasks constructed using these benchmarks. The first column indicates task IDs and the second column indicates task names. The prefix attached to some benchmark names indicates the data set size used. The third column shows the (locked) WCETs of tasks, calculated assuming that the cache sets indicated in the last column are locked. We first show detailed

Table 9 Tasks from the DSPStone Benchmark Suite

i	**Task Name**	C_i^{locked}	**Locked Sets**
1	1000fir	121667	0-250
2	1000lms	226196	100-350
3	200n_real_updates	45558	200-299
4	matrix1	90956	300-337
5	1000convolution	82571	400-649
6	300convolution	25171	500-574
7	400n_real_updates	90158	769-919
8	500fir	61167	1911-2036
9	200convolution	8992	800-849
10	300n_real_updates	67858	825-974
11	400convolution	33371	925-1024
12	600convolution	49771	652-802
13	500convolution	41571	1175-1299
14	500n_real_updates	112458	1275-1399
15	500lms	113696	1375-1500
16	600fir	73267	1000-1250
17	600lms	136196	1200-1500
18	700convolution	57971	700-900
19	700fir	85367	100-350
20	700lms	158696	200-450
21	lms	23696	100-350
22	convolution	8771	200-325

simulation results for one task set constructed using a subset of benchmarks shown in Table 9. Table 10 shows the characteristics of this task set. The first/third columns indicate task IDs and the second/fourth columns show task periods. The results of the partitioning stage of our algorithm for the above task set are shown in Table 11. The first, second and third columns show the core ID, task IDs and core utilization, respectively. In the second stage of our algorithm, we allocate the remaining tasks that cannot be partitioned onto any core as migrating tasks. Table 12 shows the allocation of portions of the migrating tasks onto cores. The first column shows the core ID. The second and third columns show task IDs and their slice numbers allocated to a given core, respectively. The fourth and fifth columns show the updated utilization and density of the cores. The last column shows the total migration overhead incurred.

We conducted a similar set of simulations for four other task sets, each containing thirteen tasks chosen from those shown in Table 9. Table 13 shows the characteristics of these four task sets. The first column shows the task set ID. The second, third and fourth columns show the set of task IDs, task periods and migrating task IDs.

Table 10 Characteristics of Task set using Real Benchmarks

i	P_i	i	P_i
1	357000	11	84000
21	60000	12	117000
3	116000	13	98000
4	204000	14	320000
5	220000	15	250000
6	60000	16	168000
7	250000	17	264000
8	142000	18	150000
9	30000	19	190000
10	178000	22	30000
2	500000	20	500000

Table 11 Non-Migrating Task Allocation: Real Task Set

c	**Task IDs**	U^c
1	17,1	0.86
2	15,14	0.81
3	19,7	0.81
4	4,5	0.82
5	16,10	0.82
6	8,18	0.82
7	12,3	0.82
8	13,21	0.82
9	6,11	0.82

Table 12 Migrating Task Allocation: Real Task Set

c	i	$slice_i$	U^c	δ^c	M_i
3	9	1	0.97	1.82	650
1	9	2	1	1	650
4	22	1	0.97	1.82	1625
5	22	2	0.97	1.82	1625

We observe that, in each of the four task sets, there are four migrating tasks. This shows that task sets that cannot be scheduled using a purely partitioned approach are schedulable using our semi-partitioned approach.

Figure 2 shows the increase in utilization and density (density is the ratio of execution time of a task to its relative deadline.) achieved by our algorithm compared to a purely partitioned approach for the four task sets shown in Table 13 and for the task set (labeled task set 5) shown in Table 10. The x-axis shows task set numbers and the y-axis shows utilization/density. Each stacked bar shows the total utilization/density of the non-migrating and migrating tasks, respectively, for a given task set. The total density of a core reflects the actual load a core is supporting when even one task on the core has a deadline less than its period. So, we choose density along with utilization to reflect the increase in work load on the core. The difference between the increase in utilization and that in density is due to the fact that migrating tasks have shorter intermediate deadlines on each core they are allocated to. As expected,

Table 13 Task Set Characteristics

Set ID	Task IDs	Periods	Migrating Tasks
Set 1	1-13	200k, 400k, 90k, 150k, 140k, 50k, 150k, 100k, 18k,110k, 66k, 80k, 80k	3,11,6,9
Set 2	7-19	150K, 100K, 18K, 110K, 66K, 100K, 80K, 200K, 190K, 120K,220K, 100K, 140K	13, 11, 9, 12
Set 3	1-7,14-20	200K, 380K, 90K, 150K, 140K, 50K, 150K, 200K, 190K, 150K, 220K, 120K, 250K	3, 6, 16, 18
Set 4	1-3, 7-13, 14-20	200K, 380K, 90K, 150K, 100K, 18K, 110K, 66K,80K, 220K, 115K, 140K, 250K	3, 11, 18, 9

our algorithm is able to achieve significantly higher utilizations compared to a purely partitioned approach.

Overall, in our simulations, we observe an average increase in utilization of 37.31% and an average increase in density of 81.36% compared to purely partitioned task allocation. This demonstrates that a semi-partitioned scheduling scheme using AAP as the underlying partitioning approach performs better than a purely partitioned approach. This improvement in performance is significant for task sets with high utilization tasks (greater than 0.5). For task sets with lower utilization tasks, the number of tasks partitioned on each core increases. Since AAP is not slack-aware, its partition could result in reduced slack times on cores.

8.2 Comparison of Semi-Partitioned Scheduling Strategies

In this section, our goal is to evaluate the performance of the approaches presented in this paper against the performance of the existing semi-partitioned scheduling algorithms proposed by Burns *et al.* (Burns et al, 2010) and Kato *et al.* (Kato et al, 2009). Our partitioning approaches, AAP and SAAP, in conjunction with our approach for scheduling migrating tasks, SBS, result in two schemes that are hereafter called SBS-AAP and SBS-SAAP, respectively.

We conduct four categories of experiments that demonstrate different characteristics of the algorithms being evaluated. The performance of an algorithm is evaluated based on the total locked utilization of tasks that the algorithm manages to schedule

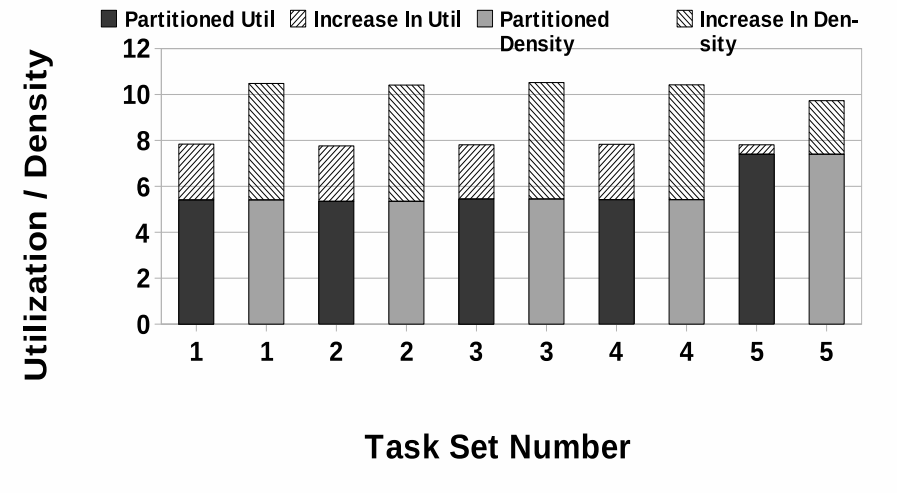

Fig. 2 Comparison of Utilization and Density for Partitioned and Semi-Partitioned Approaches

successfully. The locked utilization of a task, considered its *base* utilization, is calculated using the locked WCET of a task, which is calculated assuming that all cache chosen lines for a given task are locked. Actual utilizations of tasks after allocation onto cores may be higher due to unlocking of some cache lines under cache conflict scenarios. In our current setup, since we use a 3x3 mesh architecture and EDF scheduling on each core, the maximum utilization that may be attained is 9. In each category of experiments, we generate synthetic task sets in several different deadline ranges. For each deadline range, we generate 50 random task sets. In all result graphs presented in the remainder of this section, the y-axis shows the locked utilization of scheduled tasks, averaged over the 50 randomly generated task sets for a given deadline range, whose upper bounds are shown on the x-axis.

8.2.1 Experiment 1: Comparison of Full Algorithms

In this category of experiments, we compare the performance of the four schemes, namely C=D, EDF-WM, SBS-AAP and SBS-SAAP. For the C=D scheme, we use the *Preselect* approach because Burns *et al.* show in their paper that this outperforms the *Continuous* strategy. The C=D and EDF-WM schemes do not prescribe any particular partitioning strategy. In this category of experiments, we use a worst-fit decreasing density algorithm for partitioning in the C=D and EDF-WM schemes. The reason is that we choose tasks for allocation in non-decreasing order of their *locked densities* (derived from their locked WCETs that was explained earlier). Using a worst-fit strategy to choose a core provides the highest likelihood of actually being able to schedule the task on a chosen core since it chooses the core with the largest utilization that can still accommodate the locked WCET of the task, thereby allowing for more WCET inflation due to potential cache conflicts.

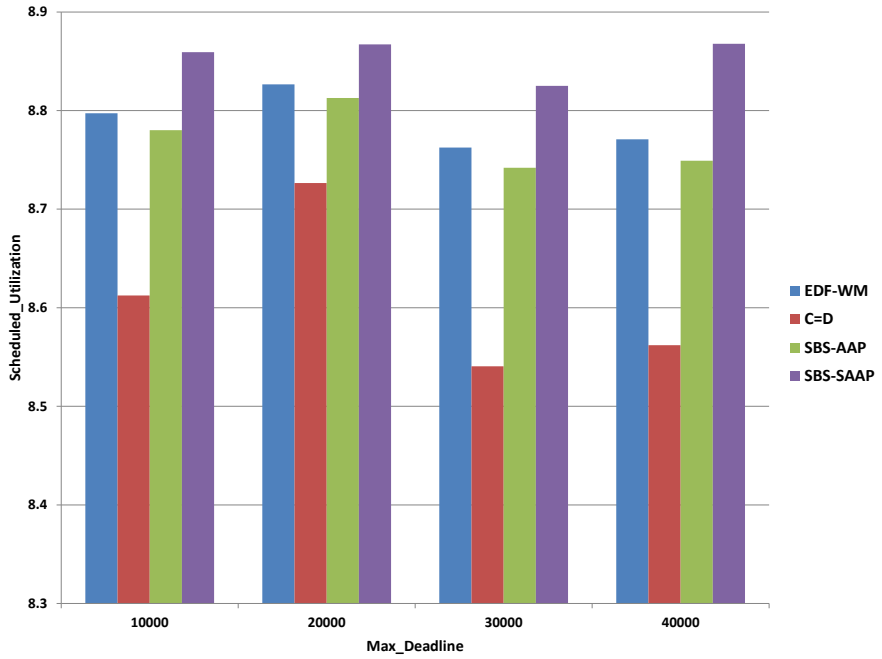

Fig. 3 Comparison of Semi-Partitioned Scheduling Algorithms

Figure 3 shows the results from this category of experiments. From Figure 3, we see that SBS-SAAP outperforms the other three algorithms, primarily because of its slack-aware partitioning strategy. Although the algorithms to calculate slack in the C=D and EDF-WM schemes are superior to SBS due to the usage of Processor Demand Analysis, SBS-AAP performs almost as well as EDF-WM in most cases and outperforms C=D. This is because of its architecture awareness and, hence, its increased tolerance to the addition of architectural overhead introduced by unlocking of cache lines. Note that C=D and EDF-WM were not designed to explicitly consider such overheads.

We observe that EDF-WM outperforms both C=D and SBS-AAP. There are two reasons for this behavior, the second among which is also the reason why SBS-AAP outperforms C=D. First, unlike C=D and SBS-AAP, EDF-WM does not insist on finding a continuous interval of time where a part of the migrating task executes non-preemptively at the highest priority. Instead, it attempts to find available utilization within a given window of time, thus increasing the likelihood of finding non-zero execution time for the migration task. Second, the Preselect approach in C=D (which is what we use in our evaluation) selects tasks with the least relative deadlines, in non-decreasing order of relative deadlines, as candidates for migration. Since this paper explicitly adds cache content migration overheads to migrating tasks, tasks with short relative deadlines are not good candidates for migration because they have less of an interval within which to accommodate migration overhead.

8.2.2 Experiment 2: Comparison with a common Partitioning Scheme

Since C=D and EDF-WM were not originally designed to explicitly consider cache-based architectures or cache content migration overheads, their performance in Experiment Set 1 is not truly indicative of the performance of their underlying strategies for scheduling migrating tasks. In order to perform a comparison on more equal ground, we now present a set of results obtained by using the architecture aware partitioning strategy, AAP, for the task partitioning phase in C=D, EDF-WM and, of course, SBS-AAP. This effectively means that, until the end of the task partitioning phase, the same task allocation decisions are made by these three schemes. SBS-SAAP still continues to use a different partitioning strategy, namely SAAP.

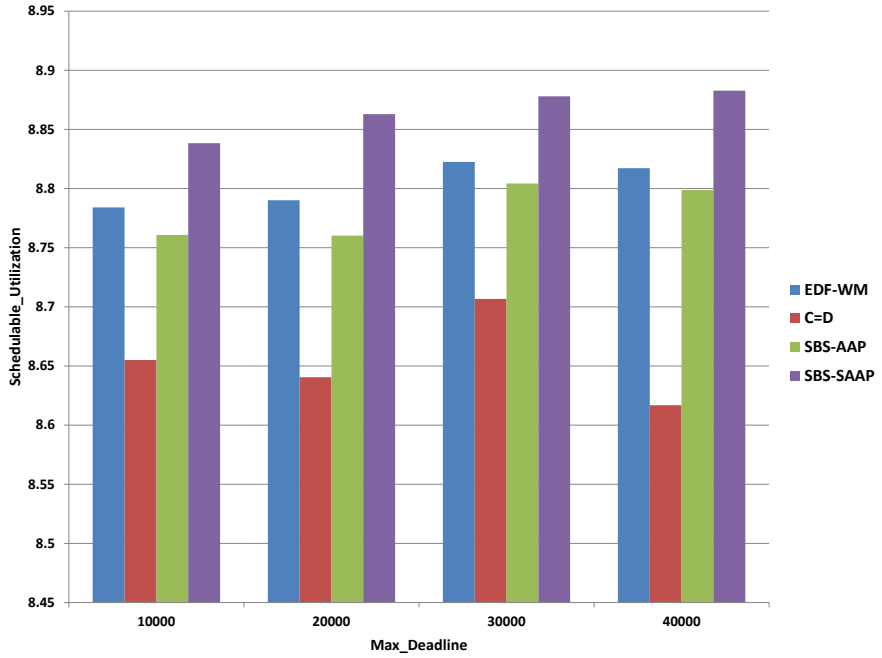

Fig. 4 Comparison with common partitioning scheme (AAP) for C=D, EDF-WM and SBS-AAP

Figure 4 shows the results of this experiment. We observe that SBS-SAAP once again outperforms the other three schemes due to its slack-aware nature. The difference in performance between EDF-WM and SBS-AAP increases (EDF-WM improves in relative performance) when compared to that in Experiment 1. This is because of the usage of a more sophisticated underlying partitioning scheme. However, we observe that C=D is stil outperformed by all other schemes. Once again, this is due to the approach used for choose migrating tasks in C=D, which prevents it from utilizing the available slack effectively. Once again, although SBS uses a simplistic slack calculation scheme, it fares well in comparison with C=D and EDF-WM.

8.2.3 Experiment 3: Comparison with common Partitioning and Task Ordering Schemes

In both Experiment Sets 1 and 2, we observed that EDF-WM and SBS-AAP have close performance, but that C=D is outperformed by both of them. As explained above, one of the reasons for the decreased performance of C=D is its approach for choosing migrating tasks. In this third category of experiments, in addition to using the same partitioning strategy for the three schemes C=D, EDF-WM and SBS-AAP, we also use the same approach and order for choosing migrating tasks. Specifically, we allocate migrating tasks in non-increasing order of utilization, as explained in Section 5.1. In effect, we purely compare the task splitting schemes of C=D, EDF-WM and SBS.

The result of this experiment is shown in graph in Figure 5. Here, we observe that, in all cases, C=D outperforms EDF-WM. This is primarily because EDF-WM does not find as much execution time for the first sub-task of a migrating task as C=D and, hence, ends up scheduling less number of migrating tasks compared to the C=D scheme. Since the current category of experiments puts EDF-WM and C=D on an even ground as far as the choice and allocation of migrating tasks is concerned, this scenario arises more often.

SBS-AAP is outperformed by both C=D and EDF-WM in this category of experiments due to its less sophisticated slack calculation scheme. However, we observe C=D and EDF-WM still fail to surpass SBS-SAAP. This is because SAAP generates more slack time as compared to AAP in the partitioning stage. So, even when it is used with the simplistic slack calculation scheme in SBS, it manages to accommodate more utilization.

8.2.4 Experiment 4: Significance of Slack-Awareness in Underlying Partitioning Scheme

In the previous experiments we observed that even the sophisticated task splitting schemes C=D and EDF-WM were outperformed by SBS-SAAP. In this section, we use SAAP as the underlying partitioning for C=D, EDF-WM and, of course, SBS-SAAP. SBS-AAP is the only scheme that continues to use AAP. We use a common task ordering algorithm for all schemes.

The results of this experiment are shown in Figure 6. Here, we observe that C=D performs better than all other schemes, including SBS-SAAP. This is because SAAP's slack-aware nature enables it to generate maximum possible slack and C=D's superior task splitting algorithm allows it to utilize this available slack more effectively. This experiment demonstrates the usefulness of a good underlying partitioning strategy in the overall effectiveness of any semi-partitioned scheduling scheme.

Although SBS-SAAP is outperformed by C=D in this experiment, we observe that the maximum difference between the utilizations is less than 0.05. We also observe that SBS-SAAP continues to outperform EDF-WM even when the latter uses SAAP as its underlying partitioning strategy. Once again, we demonstrate that, with a good underlying partitioning strategy, even a simple and computationally much less complex algorithm such as SBS performs well.

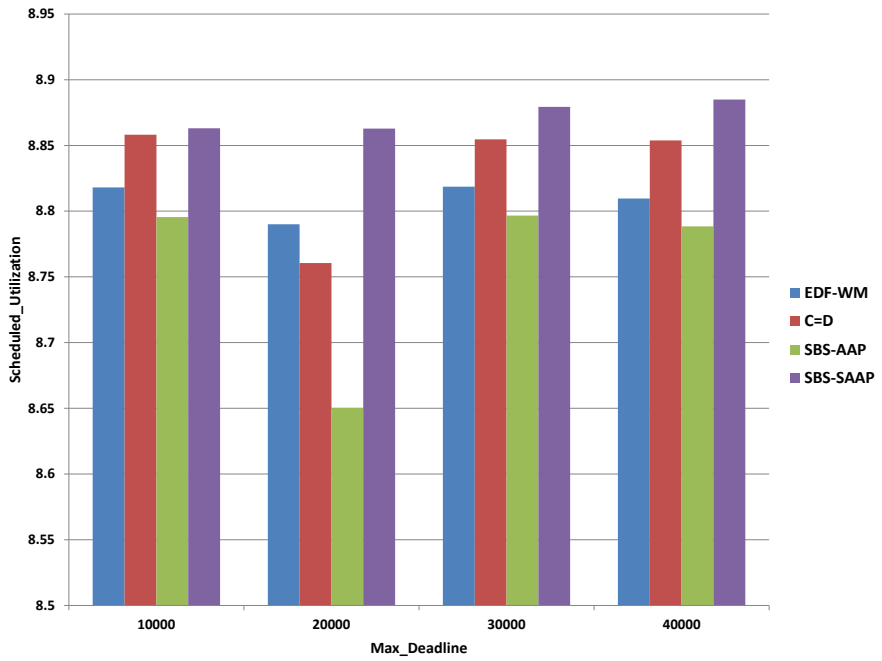

Fig. 5 Comparison with common task ordering scheme and with AAP for C=D, EDF-WM and SBS-AAP

9 Conclusion

Real-time/embedded systems are demanding increasing amounts of computational power that can only be satisfied by the use of multicore architectures. In such systems, providing *a-priori* schedulability guarantees is paramount, even in the presence of task migration among cores on systems with architectural features such as caches and shared on-chip communication infrastructure.

In this paper, we present semi-partitioned scheduling algorithms that, used in conjunction with cache locking and locked cache migration, offers a concrete and practical approach towards achieving real-time guarantees on multicore architectures without severe degradation in schedulable utilization. As part of the semi-partitioned scheduling algorithms, we present two task ordering approaches (utilization based and migration characteristics based), two task allocation approaches (Architecture Aware Partitioning or AAP and Slack and Architecture Aware Partitioning or SAAP) and a simple task splitting algorithm (SBS).

We evaluate the effectiveness of our semi-partitioned scheduling algorithm using a utilization based ordering scheme and AAP to a purely partitioned approach and, in our simulations, we observe an average increase in utilization of 37.31% and an average increase in density of 81.36% compared to a purely partitioned algorithm.

We also enhance two existing semi-partitioned scheduling schemes (C=D and EDF-WM) to explicitly consider cache content migration overhead and evaluate them

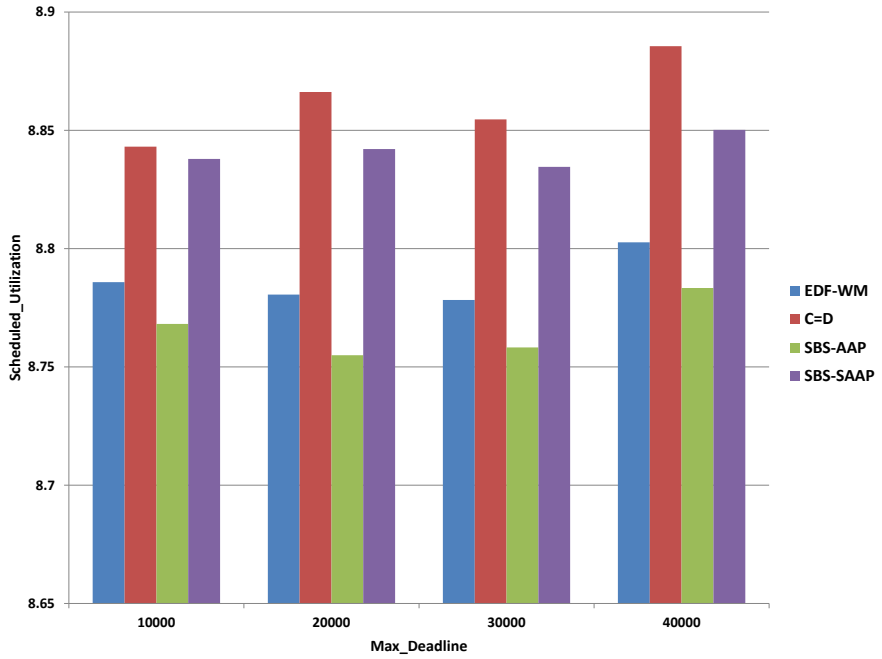

Fig. 6 Comparison with common task ordering scheme and with SAAP for C=D, EDF-WM and SBS-SAAP

alongside the schemes proposed in this paper. Although C=D and EDF-WM use more sophisticated task splitting schemes, thus providing more potential for slack utilization to schedule split (migrating) tasks, we demonstrate that their effectiveness may be reduced on a cache-based multi-core platform with a Network-on-Chip communication infrastructure unless those aspects are explicitly considered. Specifically, we demonstrate that a naive partitioning approach could decrease the effectiveness of even a sophisticated task splitting scheme such as C=D. On the other hand, an architecture and slack-aware partitioning scheme such as SAAP goes a long way in improving the overall effectiveness of a semi-partitioned scheduling scheme, even when used with a simplistic and computationally much less complex task splitting scheme such as SBS. In our simulations, the largest observed difference in the scheduled utilization when using SBS-SAAP and C=D with SAAP was less than 0.05.

References

Anderson J, Srinivasan A (2000) Early-release fair scheduling. In: Euromicro Conference on Real-Time Systems, pp 35–43

Anderson J, Bastoni A, Brandenburg B (2011) Is semi-partitioning practical ? In: Euromicro Conference on Real-Time Systems, pp 122–135

Andersson B, Bletsas K (2008) Sporadic multiprocessor scheduling with few preemptions. In: Proceedings of the 2008 Euromicro Conference on Real-Time Systems, ECRTS '08, pp 243–252

Baruah S (2007) Techniques for multiprocessor global schedulability analysis. In: IEEE Real-Time Systems Symposium, pp 119–128

Baruah S, Cohen N, Plaxton C, Varvel D (1996) Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15:600–625

Bini E, GCButtazzo (2005) Measuring the performance of schedulability tests. RTS 30(2):129–154

van den Brand J, Ciordas C, Goossens K, Basten T (2007) Congestion-controlled best-effort communication for networks-on-chip. Proceedings of the conference on Design, automation and test in Europe

Burchard A, Liebeherr J, Oh Y, Son S (1995) New strategies for assigning real-time tasks to multiprocessor systems. IEEE Trans on Computers 44(12):1429–1442

Burns A, Davis R, PWang, Zhang F (2010) Partitioned edf scheduling for multiprocessor using c=d task splitting scheme. In: Real Time and Network Systems

Chou CL, Marculescu R (2008) Contention aware application mapping for network-on-chip communication architecutres. International Conference on Computer Design - ICCD pp 164–169

Dhall S, Liu C (1978) On a real-time scheduling problem. Operations Research 26(1):127–140

Dorin F, Yomsi PM, Goossens J, Richard P (2010) Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. CoRR abs/1006.2637

Goossens K, Dielissen J, Radulescu A (2005) Aethereal network on chip: Concepts, architectures, and implementations. IEEE Design and Test 22:414421

Hoang H, Buttazzo G, Jonsson M, Karlsson S (2006) Computing the minimum edf feasible deadline in periodic systems. In: IEEE International Conference on Embedded and Real-Time Computing Systems and Application, pp 125134,

Intel (2012) Intel's single-chip cloud computer. Techresearch.intel.com/ProjectDetails.aspx?Id=1

Kato S, Yamasaki N (2008) Portioned edf-based scheduling on multiprocessors. In: Proceedings of the 8th ACM international conference on Embedded software, pp 139–148

Kato S, Yamasaki N, Ishikawa Y (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: Euromicro Conference on Real-Time Systems

Lisper B, Vera X (2003) Data cache locking for higher program predictability. In: ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp 272–282

Livani MA, Kaiser J, Jia W (1999) Scheduling hard and soft real-time communication in a controller area network. In: IFAC/IFIP Workshop on Real-Time Programming

Moir M, Ramamurthy S (1999) Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In: IEEE Real-Time Systems Symposium, pp 294–303

Murali S, Micheli GD (2004) Bandwidth-constrained mapping of cores onto noc architectures. Design, Automation and Test in Europe Conference and Exhibition pp 896 – 901

Puaut I (2006) Wcet-centric software-controlled instruction caches for hard real-time systems. In: ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, pp 217–226, DOI http://dx.doi.org/10.1109/ECRTS.2006.32

Puaut I, Decotigny D (2002) Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In: In IEEE Real-Time Systems Symposium, pp 114–123

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

Ramaprasad H, Mueller F (2010) Tightening the bounds on feasible preemptions. ACM Transactions on Embedded Computing Systems 10:27:1–27:34

Rhee CE, Jeong HY, Ha S (2004) Many-to-many core-switch mapping in 2-d mesh noc architectures. Computer Design: IEEE International Conference on VLSI in Computers and Processors,ICCD pp 438 – 443

RIDavis, ABurns (2011) Improved priority assignment for global fixed priority pre-emptive scheduling in multi-processor real-time systems. RTS 47(1):1–40

Ripoll I, Crespo A, Mok A (1996) Improvement in feasibilty testing for real-time tasks. Journal of Real-Time Systems 11(1):1939

Sarkar A, Mueller F, Ramaprasad H, Mohan S (2009) Push-assisted migration of real-time tasks in multi-core processors. In: ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems, pp 80–89

Sarkar A, Mueller F, Ramaprasad H (2011) Predictable task migration for locked caches in multi-core systems. In: ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems, pp 131–140

Shekhar M, Sarkar A, Ramaprasad H, Mueller F (2012) Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In: Euromicro Conference on Real-Time Systems, pp 331–340

Shi Z, Burns A (2008) Edzl scheduling analysis. In: Second ACM/IEEE International Symposium on Networks-on-Chip, pp 161–170

Shi Z, Burns A (2010) Schedulability analysis and task mapping for real-time on-chip communication. Real-Time Systems 46:360–385

Srinivasan A, Anderson J (2002) Optimal rate-based scheduling on multiprocessors. In: ACM Symposium on Theory of Computing, pp 189–198

Suhendra V, Mitra T (2008) Exploring locking & partitioning for predictable shared caches on multi-cores. In: Design Automation Conference, ACM, New York, NY, USA, pp 300–303, DOI 10.1145/1391469.1391545, URL http://portal.acm.org/citation.cfm?id=1391469.1391545

Tilera (2012) Tilera processor family. Http://www.tilera.com/

Zhang F, Burns A (2008) Schedulability analysis for real-time systems with edf scheduling. IEEE Transaction on Computers 58(9):12501258

Zivojnovic V, Velarde J, Schlager C, Meyr H (1994) Dspstone: A dsp-oriented benchmarking methodology. In: Signal Processing Applications and Technology