Semi-Partitioned Hard-Real-Time Scheduling Under Locked Cache Migration in Multicore Systems

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

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 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.

In this paper, we present a predictable semi-partitioned strategy for scheduling a set of independent hard-real-time tasks on homogeneous multicore platforms using cache locking and locked cache migration. Semi-partitioned scheduling strategies form a middle ground between the two extreme approaches, namely global and partitioned scheduling. 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 of task sets may be improved.

Simulation results demonstrate the effectiveness of our approach in improving task set schedulability over purely partitioned approaches while maintaining real-time predictability of migrating tasks. In our simulations, we achieve an average increase in utilization of 37.31% and an average increase in density of 81.36% compared to purely partitioned task allocation.

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 deploying multicore architectures. Modern 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, 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, providing predictable task migration is very challenging.

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

On the other hand, in partitioned scheduling, tasks are statically partitioned onto cores and remain there throughout their lifetime. A local scheduler then schedules tasks on that core according to a given 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 an NP hard problem, 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.

In this paper, we present a semi-partitioned approach for predictably scheduling periodic hard-real-time tasks using cache migration on a Network-on-Chip (NoC) based multicore architecture. As many tasks as possible are statically partitioned onto cores to minimize online cache migration overhead. The remaining tasks are allowed to migrate in a predetermined manner among a preselected subset of cores to improve task set schedulability. On each core, tasks are scheduled using an Earliest Deadline First (EDF) policy since it maximizes core utilization bound.

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 lines belonging to the task are migrated 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.

2. Background and Related Work

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

2.1. 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 [16], [15] and cache locking techniques that provide performance comparable with scratchpad-based techniques [17]. Lisper *et al.* have proposed techniques for locking data caches [11]. Recently, cache locking techniques for multicore systems with shared L2 caches have been proposed by Suhendra *et al.* [24].

2.2. Real-Time Execution 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* [7], [5], tasks are assigned to cores statically and are not allowed to migrate between cores. 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 [4], [13], [1], [23], [3]. However, the number of task migrations introduced in such schemes could be prohibitive in the context of real-time systems due online migration overheads that change the timing behavior of tasks, thus affecting the overall timing predictability of the system.

In recent work, semi-partitioned approaches have been explored to reduce the number of migrations [10], [2], [8]. Although these approaches significantly reduce the number of task migrations, 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 systems, the calculation of migration overheads is not trivial. In our work, we explicitly model cache migration and calculate the cost of migrating cache lines over the communication infrastructure.

Sarkar *et al.* have proposed proactive, push-based migration mechanisms for bus-based multicore architectures [22] and mechanisms to support migration of locked cache lines among cores [20]. In our current paper, we adopt such a push-based migration mechanism. In more recent work, Sarkar *et al.* have proposed algorithms for statically partitioning tasks that use cache locking on multicore architectures [21]. In contrast, our paper employs a semi-partitioned scheduling scheme to improve schedulability.

2.3. 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 [6], [14], [19], [26]. 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 [25], [9]. In order to support predictable real-time execution, we employ a time-division-multiplexed (TDM) approach for arbitrating core memory requests across the NoC.

3. Assumptions

Architectural and Task Model. We assume a homogeneous multicore architecture where each core has private, set associative, lockable caches. We assume 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 [25] that has five independent mesh-based NoC interconnects. We also assume support for message prioritization similar to that provided by the CAN bus protocol [12]. We assume that each core's router has a buffer with size at least equal to the size of one cache line. We assume 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 periodic hard-real-time task model with relative deadlines of tasks being less than or equal to their periods. Task characteristics are represented by the tuple (P_i, C_i, D_i), where P_i is the period, C_i is the WCET and D_i is the relative deadline of task T_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 [18] is used to calculate the worst-case execution time (WCET) of the task with a chosen set

of cache lines locked (C_i^{locked}). [1] We assume that partitioned tasks may use at most $k-1$ ways of the k available ways in a core's private cache and that each such task may lock at most one way in a given cache set. We assume that the k^{th} way is dedicated for migrating tasks. Currently, we also assume that only one migrating task may be allocated to any given core. This is done in order to avoid the possibility of contention among migration traffic when two or more migrating tasks are allocated to the same core in an effort to minimize migration overheads.

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.

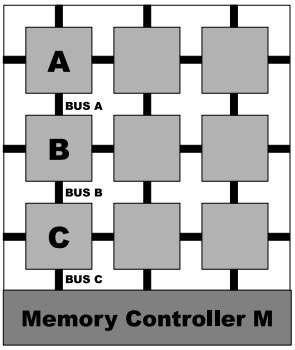

Fig. 1. Memory Traffic Routing

Bus C conveys traffic from cores A, B and C, Bus B, that from 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 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 core 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, that for traffic from core B 6 cycles and that for traffic from core C 6 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 independent of the location of the core on the mesh, thus making the worst-case execution time (WCET) independent of the physical location of the core on which it is allocated. This enables us to perform task allocation onto virtual cores and then place cores that contain different portions of a given migrating task physically close to each other in an effort to decrease the 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, is easily extends to this case.

1. 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.

4. Methodology

In this section, we describe our task allocation strategy in detail. The algorithm consists of two main steps. 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 onto a set of cores. A migrating task executes on each core it is allocated to for a prescribed amount of time and in a prescribed order. Algorithm 1 depicts these steps.

We explain our algorithm with the help of a simple example task set whose characteristics are shown in Table 1. The first column shows the task ID and the second column shows the period of the task. 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.

i	P_i	C_i^{locked}	u_i	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

TABLE 1. Example Task Set

4.1. Allocation of Tasks by Partitioning

Algorithm. Our partitioning algorithm (Lines 9 - 19 in Algorithm 1) sorts tasks in decreasing order of their (locked) utilizations and tries to allocate them one by one onto cores using a best-fit strategy. When a new task is allocated onto 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 its regions to resolve the conflicts, thereby

Algorithm 1 Semi-Partitioned Task Allocation Algorithm

```
1: Task_Allocator(Tasks, Cores)
2: Partition_Tasks(Tasks, Cores)
3: if (Tasks not Empty) then
4:    schedulable ←
5:     Allocate_Migrating_Tasks(Tasks, Cores)
6: end if
7:
8: Partition_Tasks(Tasks, Cores)
9: while (Tasks not Empty) do
10:   Task ← Max_Util_Task(Tasks)
11:   Core ← Min_Util_Change_Core(Task, Cores)
12:   if (Util(Core) + Util(Task) ≤ 1) then
13:      Allocate_Task_To_Core(Task, Core)
14:      Delete_Task(Tasks, Task)
15:   else
16:      Add_Task(Migrate_List,Task)
17:      Delete_Task(Tasks, Task)
18:   end if
19: end while
20:
21: Allocate_Migrating_Tasks(Migrate_List, Cores)
22: while (Migrate_List not Empty) do
23:   Task ← Max_Util_Task(Migrate_List)
24:   Done ← false
25:   while (Done ≠ true) do
26:      Core ← Find_Max_Slack_Core(Cores)
27:      if (Valid(Core)) then
28:         Allocate_Task_To_Core(Task, Core)
29:         Delete_Core(Cores, Core)
30:      else
31:         return false
32:      end if
33:      Mig_Curr ← Calc_Curr_Migration_Overhead(Task)
34:      u_Task ← Calc_Rem_Util(Task, Mig_Curr)
35:      Mig_Last ← Calc_Last_Migration_Overhead(Task)
36:      if (u_Task + Mig_Last ≤
         (1 - Max_Util_Core(Cores))) then
37:         Core ← Min_Util_Min_Slack_Core(Cores, Task)
38:         if (Valid(Core)) then
39:            Allocate_Task_To_Core(Task, Core)
40:            Delete_Task(Migrate_List, Task)
41:            Delete_Core(Cores, Core)
42:            Done ← true
43:         else
44:            return false
45:         end if
46:      end if
47:   end while
48: end while
49: return true
```

increasing the utilization of these tasks. In such a situation, our algorithm chooses to unlock conflicting regions with minimum access frequency.

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 (locating independent) memory access latency for each line is calculated using the NoC routing and arbitration model presented in Section 3. The new task is allocated to the core that suffers the minimum change in utilization due to the addition of the new task, including both the utilization of the new task itself and the change in utilizations of existing tasks on the core, if any, due to region unlocking. If two or more cores have the same change in utilization, the core with the least absolute utilization among them is chosen. If the new task cannot be accommodated on any core due to utilization bounds, it becomes a candidate for migration.

Example. In the example shown in Table 1, 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 it cannot be accommodated on any of the cores. Hence, this task is considered for execution as a migrating task.

Justification of Partitioning Heuristics. Tasks are allocated in decreasing order of utilizations in order to maximize the chances of high utilization tasks being partitioned. By doing so, our algorithm retains lower utilization tasks as candidates for migration, potentially resulting in lower online migration overhead and more room for accommodating this overhead.

When allocating a new task to a core that already contains tasks, our algorithm chooses the core that suffers the minimum change in utilization due to the addition of the new task. This is 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.

When there are two or more cores on which addition of a new task leads to the same change in utilization, our algorithm chooses the core that has minimum absolute utilization among them. This is done to improve load balancing and, hence, minimize thermal degradation of cores.

4.2. Allocation of Migrating Tasks onto a Set of Cores

The subset of tasks that cannot be accommodated by the partitioned approach presented above 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 1.

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

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 $\max(D_{min}^c - \sum_{j \in p(c)} C_j^c, 0)$.

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

$$d_{non-mig} < t_{mig} + \Delta_{mig}^c \quad (2)$$

$$e_{non-mig}^{left} > 0 \quad (3)$$

From Equation 1, 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 $d_{non-mig} - (t_{mig} + e_{non-mig}^{left})$. From Equation 2, we get $\Delta_{non-mig} < \Delta_{mig}^c - e_{non-mig}^{left}$ From Equation 3, $\Delta_{non-mig} < \Delta_{mig}^c$. This is a contradiction since $\Delta_{non-mig} \geq \Delta_{mig}^c$.

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_{mig}^c 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 22 - 49 of Algorithm 1).

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

Step 2. Allocate Task Portion. Cores are sorted in decreasing 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 only one migrating task may be allocated to a given core according to our current assumption. These steps are shown in Lines 26 - 32 of Algorithm 1. The relative deadline of this portion of the migrating task is set to be equal to the available slack time and the migrating task is guaranteed to execute at the highest priority on this core without violating schedulability of non-migrating tasks on the core in accordance with Theorem 1. 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. This is in contrast to other semi-partitioned scheduling approaches that require modifications to the scheduling policy and the schedulability analysis [10].

Step 3. Calculate Remaining Task Utilization. The remaining WCET of the migrating task, including the overhead of migration is calculated using Equation 4 (Lines 33 - 34 of Algorithm 1).

$$C_i^{rem,m} = C_i^{rem,m-1} - \max_{c \in cores} \Delta_i^c + M_i^m \quad (4)$$

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 m^{th} migration, calculated using Equation 5.

$$M_i^m = (Read + Write + \sum_{h=1}^{nh(src^m, dst^m)} l_h) * nl_i^{migrated} \quad (5)$$

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 35 - 36 of Algorithm 1), 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 3, 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).

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 37 - 42 of Algorithm 1. 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.

Example. Step 1. Table 2 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.

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

TABLE 2. Core Slack Times After Partitioning

Step 2: 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 3: 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 1, 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 4. We then check whether 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 possible cores.

Step 4: 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.

4.3. Discussion

4.3.1. Applicability of our Algorithm. Our method is particularly suited to task sets in which several tasks have high utilizations (≥ 0.5) and a few tasks have lower utilizations. Due to 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. Furthermore, we explicitly allow consideration of different utilization caps for cores.

4.3.2. Comparison with Existing Work. An advantage of our algorithm for allocating migrating tasks compared to a semi-partitioned approach proposed in related work [8] 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. Another related work [10] 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. On the other hand, our algorithm uses the EDF policy and the corresponding schedulability test.

In our work, we specifically target locked-cache-based systems and strive to maximize locking while minimizing migration overheads. In contrast, existing techniques assume a cacheless system and, hence, do not explicitly consider migration overheads. This results in a significantly higher baseline for existing techniques in terms of system utilization and schedulability, thus preventing a fair comparison.

4.3.3. Algorithm Complexity. The partitioning stage of our algorithm iterates over tasks, and, for each task, iterates over cores to check for possible allocations. Similarly, to allocate migrating tasks onto cores, our algorithm iterates over migrating tasks and, for each task, could allocate portions of the task on every core in the system in the worst-case. Hence, the overall time complexity of our algorithm is $O(n * m)$, which is acceptable since the algorithm is offline.

5. Simulation Setup

We have designed and implemented a software simulator for the algorithm presented in Section 4. The architectural configuration for our simulations is shown in Table 3. The external memory 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 3), **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). We assume that all four ways of the L1 data cache are lockable by tasks. However, one lockable way is reserved for migrating tasks. 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.

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**

TABLE 3. System Configuration

6. Simulation Results

In this section, we present our simulation results. In all our simulations, tasks are assumed to have relative deadlines equal to their periods and phases of zero (synchronous task set). We conducted simulations with benchmarks from the DSPStone benchmark suite [27]. WCET values for tasks constructed with these benchmarks were calculated using a static timing analysis framework developed in prior work [18]. We also conducted simulations with synthetically generated tasks.

Table 4 shows details of tasks constructed using benchmarks from the DSPStone suite. 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 for the task. The third column shows the locked WCETs of tasks and the last column shows their locked cache sets. We first show detailed simulation results for one task set constructed using a subset of benchmarks shown in Table 4. Table 5 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 6. 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 migrating tasks onto cores. Table 7 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 4. Table 8 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

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

TABLE 4. Tasks from the DSPStone Benchmark Suite

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

TABLE 5. Characteristics of Task set using Real Benchmarks

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 6. Non-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

TABLE 7. Migrating Task Allocation: Real Task Set

migrating task IDs. In this set of simulations, we show that the task sets that cannot be scheduled using a purely partitioned approach are schedulable using our algorithm. In each of the four task sets, there are four migrating tasks. Although we use reasonably small task sets, we believe that it suffices to demonstrate the utilization benefit obtained by using our algorithm compared to a 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 8 and for the task set (labeled task set 5) shown in Table 5. 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 the migrating tasks have shorter intermediate deadlines on each core they are allocated to. As expected, 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.

In practice, it is sometimes useful, from a power/energy consumption standpoint, to run cores below 100% load. We demonstrate the effectiveness of our algorithm under different utilization caps for individual cores, namely 0.5, 0.75 and 1.

Table 9 shows task set characteristics for a set of simulations where task sets are constructed using benchmarks from the list

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

TABLE 8. Task Set Characteristics

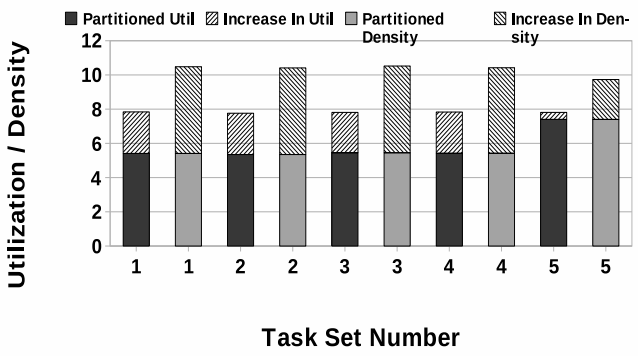

Fig. 2. Comparison of Utilization and Density for Partitioned and Semi-Partitioned Approaches

shown in Table 4 for different core utilization caps. Figure 3 shows the results obtained for this set of simulations. The x-axis shows task sets, each with a different utilization cap, and the y-axis shows the utilization of these task sets under our algorithm and the increase in utilization required in order to schedule the same task sets using a partitioned approach. For example, task set 2 needs a minimum of 0.85 utilization cap on each core to completely partition the tasks onto cores. Using our semi-partitioned approach, the same task set is schedulable under a utilization cap of 0.75 for each core. So if we chose to execute the cores uniformly on a lower load than partitioned would allow, then we can use our algorithm to schedule tasks

Utilization Cap	0.5	0.75	1
Set ID	1	2	3
Task IDs	1-13	1-13	1-13
Periods	400k, 750k, 150k, 300k, 275k, 100k, 300k, 200k, 30k, 225k, 116k, 150k, 140k	240K, 450K, 150K, 180K, 160K, 80K, 180K, 120K, 30K, 130K,65K, 100K, 140K	200K, 400K, 90K, 150K, 140K, 50K, 150K, 100K, 18K, 110K, 66K, 80K, 80K
Migrating Tasks	5,13,6,9	6, 3, 9, 13	3, 11, 6, 9

TABLE 9. Task Set Characteristics under Utilization Caps

onto cores.

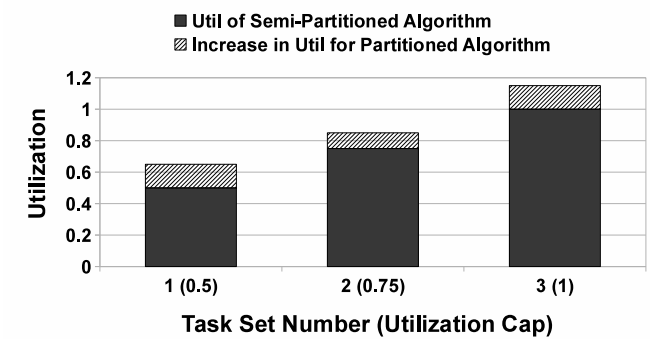

Fig. 3. Minimum Required Core Utilization Caps for Partitioned and Semi-Partitioned Approaches

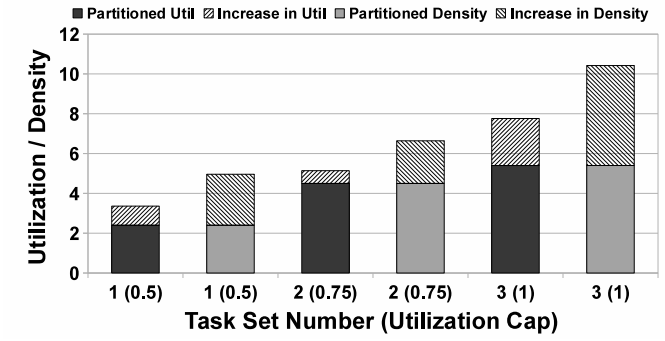

Fig. 4. Comparison of Utilization and Density for Partitioned and Semi-Partitioned Approaches under Core Utilization Caps

Figure 4 shows the results of a set of simulations using synthetically constructed task sets, under core utilization caps of 0.5, 0.75 and 1, respectively. The x-axis shows task sets and the y-axis shows the utilization/density achieved, with each stacked bar showing total utilization/density for non-migrating and migrating tasks, respectively. Once again, this set of simulations show that, under the same utilization caps, our algorithm allows significantly higher utilization and density compared to a partitioned approach.

7. Conclusions

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 a semi-partitioned scheduling strategy that, 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. We demonstrate the effectiveness of our approach compared to a purely partitioned approach using software simulations.

References

[1] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[2] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 243–252, 2008.

[3] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 119–128, 2007.

[4] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

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

[6] C.-L. Chou and R. Marculescu. Contention aware application mapping for network-on-chip communication architecutres. *International Conference on Computer Design - ICCD*, pages 164–169, 2008.

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

[8] F. Dorin, P. M. Yomsi, J. Goossens, and P. Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. *CoRR*, abs/1006.2637, 2010.

[9] Intel's single-chip cloud computer. techresearch.intel.com/ProjectDetails.aspx?Id=1.

[10] S. Kato and N. Yamasaki. Portioned edf-based scheduling on multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 139–148, 2008.

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

[12] M. A. Livani, J. Kaiser, and W. Jia. Scheduling hard and soft real-time communication in a controller area network. In *IFAC/IFIP Workshop on Real-Time Programming*, 1999.

[13] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *IEEE Real-Time Systems Symposium*, pages 294–303, Dec. 1999.

[14] S. Murali and G. D. Micheli. Bandwidth-constrained mapping of cores onto noc architectures. *Design, Automation and Test in Europe Conference and Exhibition*, pages 896 – 901, 2004.

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

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

[17] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation and Test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[18] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 10:27:1–27:34, 2010.

[19] C.-E. Rhee, H.-Y. Jeong, and S. Ha. Many-to-many core-switch mapping in 2-d mesh noc architectures. *Computer Design: IEEE International Conference on VLSI in Computers and Processors,ICCD*, pages 438 – 443, 2004.

[20] A. Sarkar, F. Mueller, and H. Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 131–140, Apr. 2011.

[21] A. Sarkar, F. Mueller, and H. Ramaprasad. Static task partitioning for locked caches in multi-core real- time systems. Technical Report TR 2011-11, Dept. of Computer Science, North Carolina State University, 2011.

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

[23] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[24] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

[25] Tilera processor family. http://www.tilera.com/.

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

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