# Static Task Partitioning for Locked Caches in Multi-Core Real-Time Systems

Abhik Sarkar, Frank Mueller
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
asarkar@ncsu.edu, mueller@cs.ncsu.edu

Harini Ramaprasad
Dept. of Electrical and Computer Engg.
Southern Illinois University Carbondale
Carbondale, IL 62901
harinir@siu.edu

## Abstract

*Massive multi-core architectures with tens of cores are becoming more prevalent in embedded systems. However, their acceptance in the real-time systems domain is rather low due to challenges in system analysis and predictability. Recent real-time systems research has focused on shared cache architectures. In such systems, tasks across all cores share the same cache (typically at level 2). In contrast, tile-based architectures with massive numbers of cores have private caches, only. At larger numbers of cores, it becomes important to utilize all resources efficiently and share cores among hard, soft and non real-time tasks. In such a scenario, it becomes difficult to analyze cache behavior statically. Consequently, hard real-time systems become subject to conservative analysis including timing bounds based on the assumption that data references default to lower levels of caches/memory. Locking cache lines in hard real-time systems is a common means to ensure timing predictability of data references and to lower bounds on worst-case execution time, especially in a multi-tasking environment. However, cache locking poses a challenge to hard real-time systems on multi-core architectures. Static scheduling on hard real-time tasks does not consider conflicts among locked regions within caches. This work proposes three static scheduling schemes as a remedy: (1) Greedy First Fit Decreasing (GFFD) and (2) Colored First Fit Decreasing (CoFFD). This work also adapts these algorithms for conflict resolution partially locked regions. Experiments indicate that CoFFD consistently outperforms GFFD for lower number of cores and lower system utilization. With partial locking, the number of cores in some cases is reduced by almost 50% with an increase in system utilization of 10%. Overall, this work is unique in considering the challenges of future multi-core architectures for real-time systems and provides key insights into task partitioning with locked caches for architectures with private caches.*

## 1. Introduction

Multi-core architectures have become prevalent in embedded system design. This is evident from the variety of multi-core processors available today, such as the 4-core MPCores from ARM, the 8-core P4080 PowerPC from Freescale and the 64-core TilePro64 from Tilera [1], which find applications in power control systems, satellites and network packet processing. However, hard real-time system designers have been skeptical in adopting these architectures. Unpredictability of multi-core caches have been a significant contributing factor to this skepticism.

Research on cache contention has primarily considered shared caches. This simplifies the problem as all tasks are considered to be contending for the shared cache space. Most contemporary research aims at optimizing the analysis on aforementioned systems [6], [9]. Such schemes become inapplicable to scalable multi-cores, such as shown in Figure 1. These architectures use private L1+L2 caches. Thus, any task allocation algorithm requires prior knowledge of each task's Worst Case Execution Time (WCET). However, the WCET of a task obtained by static cache analysis depends on cache analysis of all other tasks on a particular core. In this work, it is assumed that private L2 caches are large enough to hold the data space and instructions of hard real-time tasks. This simplifies the analysis of L2 caches as any access to the L2 cache is a hit after a compulsory miss on warm-up. Thus, a tighter upper-bound on the Worst Case Execution Time (WCET) can be established by modeling references resolved at the L2 level as hits after the warm-up phase of the first job execution in a periodic task system. Still, the access latency of L2 caches is an order of magnitude higher than that of L1 caches so that bounds on WCET are not as tight as they could be. To further tighten WCET bounds, cache locking of selected lines in L1 can be employed on scalable multi-core platforms.

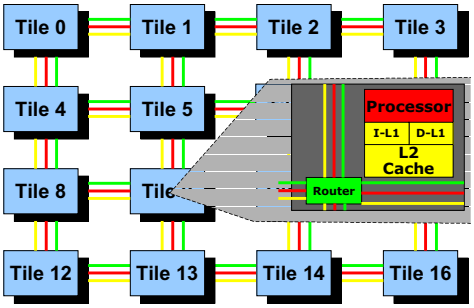


Fig. 1. Tile-based Architecture

In general, cache locking techniques provide predictability to a task's cache access behavior. Cache locking can be realized at various granularities. Studies on uni-processor cache locking have assumed the entire L1 cache to be locked [16], [17]. Another study on cache locking for shared caches has assumed locking individual cache lines [18]. Locked caches on uni-processors identify sets within a single cache way for a given task set to improve predictability and indirectly utilization/response time of tasks while ensuring schedulability on a single core. In contrast, our work extends to scalable

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

multi-core architectures where tasks are statically partitioned. Our work focuses on distributing tasks over disjoint cores while considering their locked state. A real-time system developer may choose to lock a set of cache lines to tighten WCET bound. This work uses these tightened WCET bounds to statically allocate tasks on disjoint set of cores.

Prior literature on uni-processor locking techniques focuses on filling a single cache way while reducing the overall utilization of a core. Reduction of the system utilization can be achieved by placing all tasks with conflicting locked cache regions on different cores. However, such a scheme would consume a large number of cores and result in under-utilization of computing resources. Also, multiple cache ways per L1 cache can be dedicated to locking. Hence, the objective of allocating tasks on scalable multi-cores has to be balanced between the following objectives:

1) Reduction of the number of cores; and
2) Reduction of the overall system utilization.

Static task partitioning has been considered as a viable scheduling option for real-time tasks on multiple cores. Such scheduling schemes aim at minimizing the number of cores for a set of tasks with given worst-case execution time (WCET). However, partitioning tasks with locked cache regions involves resolving the conflicts between locked regions of different tasks.

In this work, we develop and evaluate two partitioning algorithms: (1) Greedy First Fit Decreasing (GFFD), and (2) Colored First Fit Decreasing (CoFFD). GFFD is a variant of the general First Fit Decreasing (FFD) algorithm that tries to allocate tasks onto a minimum number of cores [3]. This scheme lacks prior information on the number of cores of a concrete processor but rather reasons abstractly about the minimum number of cores of a hypothetical processor design. CoFFD, a more sophisticated scheme, exhibits a novel approach based on graph coloring that delivers task partitioning. In contrast to GFFD, CoFFD initially assumes a given number of cores for an architecture. The algorithm then tries to allocate a given task-set onto the fixed number of cores. In case of failure, the number of cores is incremented and the attempt to allocate tasks to cores is repeated. If the objective is to achieve minimum utilization, tasks should be allocated with all candidate regions locked as this lowers their WCET. Table 1 depicts a comparison of the number of allocated cores for different task-sets using GFFD and CoFFD on an architecture that has a direct-mapped locked L1 data cache.

TABLE 1. CoFFD vs GFFD: Locked WCET

| Number of Tasks | System Utilization | Number of Cores | |
|---|---|---|---|
| | | GFFD | CoFFD |
| 42 | 20.25 | 27 | 25 |
| 42 | 13.42 | 25 | 24 |
| 42 | 9.39 | 24 | 23 |

The first and second columns depict the number of tasks in the task-set and the utilization of the task-set, respectively. The third and the fourth columns show the number of cores consumed by the task-set when using GFFD and CoFFD, respectively. These results indicate that CoFFD consistently delivers a partition with fewer number of cores. If the objective is to minimize the number of cores, the two algorithms are adapted to consider two different WCETs, one with locking all the regions specified by the developer and one without locking any of those regions for every task. The algorithms select one of these versions to avoid lock conflicts while ensuring that utilization constraints are met. We observe that CoFFD consistently results in allocating fewer cores. Given a task utilization $u_i$ of task $i$, task-sets composed of high utilization tasks $(0.55 > u_i > 0.40)$ allocate fewer cores under CoFFD with at most 3% higher system utilization than GFFD. For low utilization tasks-sets $(0.33 > u_i > 0.15)$, CoFFD allocates fewer cores and lowers system utilization by up to 40% over GFFD.

We also propose a mechanism to resize lock regions so that they become partially unlocked. This scheme is applicable when the programmer can accurately provide the number of references to a locked cache line. The two algorithms, GFFD and CoFFD, were adapted to exploit this per-line reference frequency information, based on which they choose whether to retain the lock of a line or unlock it due to lock conflicts of lines between disjoint tasks. We observe that such a mechanism can further reduce the number of allocated cores. It may even allow GFFD to perform at par with CoFFD. Overall, we provide key insights into task partitioning with locked caches for large-scale multi-core architectures with private caches.

**Summary of contributions:** This research makes the following contributions in the context of hard real-time systems with cache locking:

1) This work is the first to employ locked caches on massive multi-core architectures for hard real-time systems.
2) We propose GFFD, an allocation scheme that partitions a given set of tasks with conflicts in their locked cache regions so that the number of allocated cores is kept low. This algorithm is further adapted to resolve conflicts by (i) unlocking entire task or (ii) resizing locked regions.
3) We propose Colored First Fit Decreasing (CoFFD) that derives task allocations for a given number of cores resulting in a feasible schedule. This algorithm is further employed to reduce the number of cores relative to Greedy First Fit Decreasing (GFFD).
4) We propose a novel mechanism that allows tasks to resolve conflicts by partially unlocking the locked regions and inflating their WCETs accordingly. This method aims at improving the schedulability of task sets on a given number of cores when resolution of conflicts by partial unlocking result in lower system utilization than unlocking an entire task.

## 2. Related Work

In past decade, there has been considerable research promoting locked caches in the context of multi-tasking real-

time systems. Static and dynamic cache locking algorithms for instruction caches have been proposed to improve system utilization in [16], [15]. Several methods have been developed to lock program data that is hard to analyze statically [19]. Further techniques have been developed for cache locking that provide comparable performance obtained on scratchpad allocation [17]. Recently, cache locking has also been proposed for multi-core systems that use shared L2 caches [18]. This trend is a strong proponent of cache locking as a viable solution in future real-time system designs on multi-cores.

Choffnes *et al.* have proposed migration policies for multicore fair-share scheduling [7]. Their technique strives to minimize migration costs while ensuring fairness among the tasks by maintaining balanced scheduling queues as new tasks are activated. Calandrino *et al.* propose scheduling techniques that account for co-schedulability of tasks with respect to cache behavior [2], [4]. Their approach is based on organizing tasks with the same period into groups of cooperating tasks. All these methods improve cache performance in soft real-time systems. Li *et al.* discuss migration policies that facilitate efficient operating system scheduling in asymmetric multicore architectures [11], [12]. Their work focuses on fault-and-migrate techniques to handle resource-related faults in heterogeneous cores and does not operate in the context of real-time systems. Eisler *et al.* [8] develop a cache capacity increasing scheme for multicores that scavenges unused neighboring cache lines.

Paolieri *et al.* [14] have proposed TDMA-based bus and L2 cache access to improve predictability on multi-core architectures. Their work focuses on supporting hard real-time applications on multi-cores but assumes shared L2 caches with contention due to accesses by different tasks. Ouyang *et al.* [13] have proposed extending Quality of Service support to mesh-based interconnects but their study is limited to the on-chip network traffic.

## 3. System Design

In this section, we describe our system architecture and assumptions to WCET analysis for this study. The objective of this work is to best utilize a private cache architecture. This corresponds to the current trend in potentially mesh or tile-based multi-core designs. Tile-based architectures consist of a large number tile processors (cores). Each tile consists of an in-order processor, a private L1, a private L2 cache and a router (see Figure 1). Each tile acts as a node on a mesh interconnect. Recent work has added Quality-of-Service (QoS) policies to mesh-interconnects [13]. We have identified these trends as the driving force for the simplification of our system. We assume an architecture that has private caches and has a QoS-based interconnect. We assume that the first level of cache allows a certain number of ways of the associative cache to be locked as shown in Figure 2. We also assume that the L2 caches are large enough with high associativity so that the address space of allocated hard real-time tasks on a core fit within the L2 cache. Thus, we assume that the off-chip references occur only while accessing sensory data, which accounts for a very small fraction of the total references. Also, these systems can have inclusive or non-inclusive L2 caches. With inclusive caches, the locked regions in L1 need to be locked in L2 as well. Our algorithms are applicable to a system considering both data and instruction caches. However, for the simplicity of analysis we assume that instruction references for hard real-time tasks are all hits at the first level of cache. We also assume that loads to the lines that have not been locked in the L1 cache bypass the L1 cache (as in a previous research work [10]). This allows cores with lower core utilization to co-schedule non-real-time tasks along with hard real-time tasks without affecting the deterministic behavior of the latter. Such hybrid execution of application tasks have been considered in recent research [14]. We assume that a hard real-time task can only lock one cache line per set. Thus, for a 8KB L1 cache with an associativity of two, a hard real-time task can lock up to 4KB of cache content.

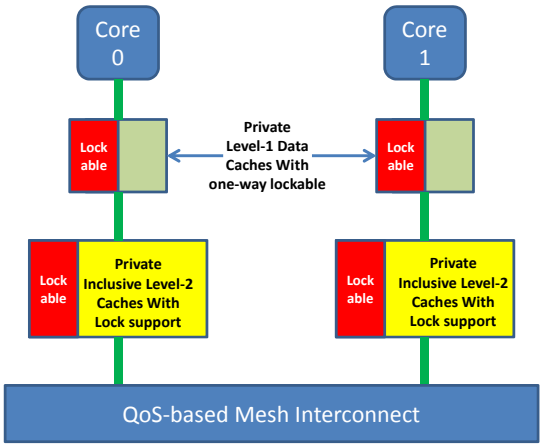


Fig. 2. A Lock-based Architecture

We assume that all hard real-time tasks are periodic. Each task's deadline is the same as its period, i.e., an invocation of a task's job has to finish before its next invocation. We further assume that the system runs a scheduler per core. Each of these schedulers independently schedules the tasks allocated to this core. We assume them to utilize Earliest Deadline First (EDF) scheduling. EDF optimally schedules tasks for uni-processor, i.e., the utilization bound for each core is defined by the following equation:

$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$, where $C_i$ and $P_i$ are the WCET and the period of the $i^{th}$ task, respectively. Deadlines are assumed to be the same as the periods.

For the algorithms, each task needs to provide the following information: $< list_{locked-sets}, WCET_{locked}, WCET_{unlocked} >$. $list_{locked-sets}$ is the list of sets where the programmer intends to lock a cache line for the task. $WCET_{locked}$ and $WCET_{unlocked}$ are the WCETs of a task when all the lines of $list_{locked-sets}$ are locked and unlocked, respectively. $WCET_{locked}$ does not include the overhead of loading the contents of a task because it is a one-time cost incurred at system start-up.

We also assume that the real-time tasks are pairwise independent. Hence, these tasks do not cause any coherence traffic on the interconnect.

## 4. Task Partition Algorithms

Static task partitioning algorithms for multi-core architectures have been widely studied. Most of these approaches consistently aim at minimizing the number of cores utilized [3]. They use bin-packing schemes considering a single utilization value per task. However, locked caches provide us with a tuple of data as discussed in the previous section. Initially, our algorithms consider two values, $WCET_{locked}$ and $WCET_{unlocked}$. In Section 4.3, we discuss a mechanism with the objective of reducing the impact of conflicts.

The $list_{locked-sets}$ item is used to deduce a conflict matrix $M_{conf}$ for locked tasks. A conflict among the locked sets indicates the existence of common locked cache set(s). Each empty entry in $M_{conf}(i, j)$ signifies the absence of conflicts between tasks $i$ and $j$ while every filled entry signifies existence of a conflict. We now present our task allocation algorithms.

### 4.1. Greedy First Fit Decreasing (GFFD)

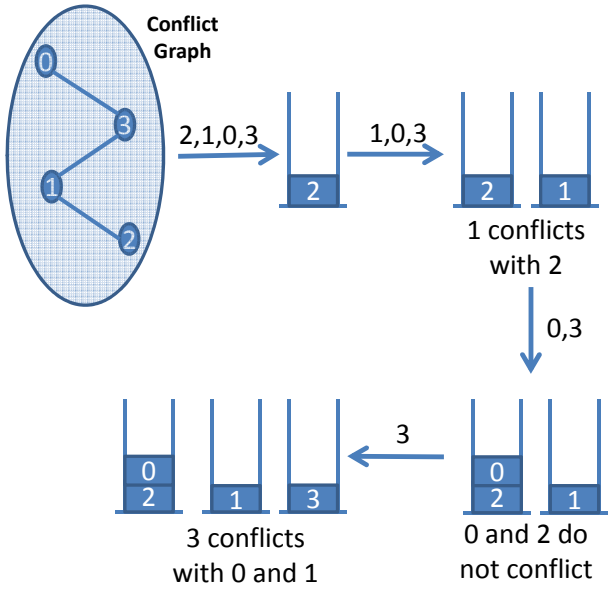

Fig. 4. First Fit Decreasing in Operation

First Fit Decreasing (FFD) task partitioning is a commonly used algorithm [3]. Figure 4 depicts an example. An undirected conflict graph of four nodes/vertices is depicted in the figure. A conflict graph in the context of task partitioning is a graph $G = (V; E)$, where every vertex/node $v \in V$ corresponds uniquely to a task and an $edge(i; j) \in E$ indicates that tasks $i$ and $j$ are in conflict and cannot be allocated onto the same core. The objective is to map nodes into buckets while keep in the number of buckets low. As such bin-packing is known to be NP-hard, heuristic approaches are generally employed. The FFD algorithm arranges nodes in traversal order via heuristics before allocating them. In this example, the algorithm establishes an allocation order of nodes 2, 1, 0 and 3. At each step, the node in question checks if it can be placed within any of the existing buckets. A node can be allocated to a bucket if the bucket does not contain any node that conflicts with it. In the example, node 0 gets allocated to a bucket that contains node 2, which does not conflict with 0. In case all buckets conflict, a new bucket is created, e.g., during the allocation of nodes 1 and 3.

**Input**: $M$ : Set of Tasks, $Assoc$ : Number of locked ways per cache, $M_{conf}$ : conflict Matrix
**Output**: $N_{procs}$ number of processors

```
1  N_procs := 1 ;
2  M.sort(decreasing u_locked);
3  while M is not empty do
4      Success := false ;
5      N_procs.sort(decreasing utilization) ;
6      i := M.front;
7      foreach N_procs j do
8          if IsAllocatable(j,i,Assoc,M_conf) ≠ −1 then
9              if i.u_locked ≤ 1 - j.u then
10                 allocate task i to core j in kth way;
11                 j.u += i.u_locked;
12                 Success := true ;
13                 break ;
               end
           end
       end
14     if Success = false then
15         foreach N_procs j do
16             if Success = false then
17                 if i.u_unlocked ≤ 1 - j.u then
18                     allocate task i to core j;
19                     j.u += i.u_unlocked;
20                     Success := true ;
21                     break;
                   end
               end
           end
       end
22     if Success = false then
23         allocate New_proc;
24         N_procs := N_procs ∪ New_proc;
25         allocate task i to New_proc;
26         New_proc.u = i.u_locked ;
       end
   end
```

**Algorithm 1:** Greedy First Fit Decreasing Heuristic (GFFD)

We implemented a modified version to the FFD algorithm. We call this Greedy First Fit Decreasing (GFFD). Algorithm 1 presents the implementation details of the algorithm. This algorithm takes a task set and the number of locked ways per cache as an input. The idea is to incrementally add cores to the

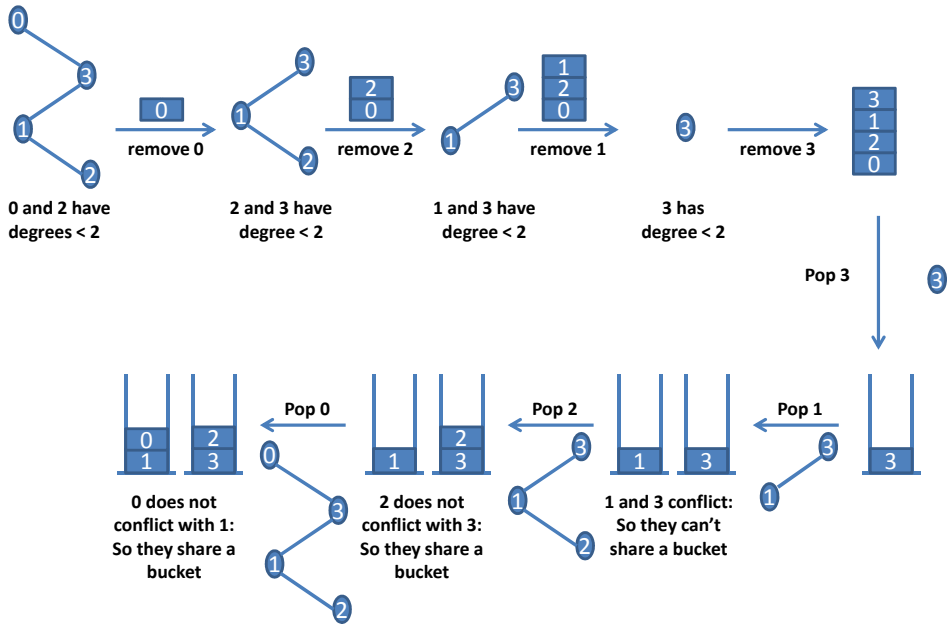


Fig. 3. Chaitin's Coloring in Operation with 2 Colors

schedule starting with an initial number of cores, $N_{procs}$, of 1. Line 2 sorts the tasks in decreasing order of their utilization under locking ($u_{locked}$). Lines 3-26 represent a loop that picks tasks in order and tries to allocate them to $N_{procs}$. At the start of each loop, the $N_{procs}$ cores are sorted in decreasing order of core utilization with the objective of maximizing the utilization of cores. The algorithm tries to allocate task $i$ to core $j$. The procedure IsAllocatable() returns the cache way that is still unassigned to any locked lines of tasks that conflict with any locked lines of task $i$. In case a valid cache way is found and the allocation of the task with the locked region passes the schedulability test, the task is allocated to the core. Allocation updates the utilization of the core and also appends the task to the list of core-allocated tasks with a specific lock-way for the associative cache. If, however, all the lockable cache-ways of the core's L1 are in conflict or the schedulability test fails, the algorithm tries to allocate the task to another core until it runs out of cores in $N_{procs}$. If the task remains unallocated, lines 14 through 26 try to allocate it without locking, i.e., with a utilization of $u_{unlocked}$. The task may fail allocation if it fails the schedulability test for each core (line 20). In such a scenario, a new core, $New_{proc}$, is allocated, which is then added to the list of cores in $N_{procs}$. The task on this new core gets allocated with $u_{locked}$. We call this algorithm greedy because it greedily allocates a task before moving on to the next task.

### 4.2. Colored First Fit Decreasing (CoFFD)

GFFD identifies task conflicts only after a task has been committed for allocation, even though a conflict matrix is already present. The algorithm does not have a prior notion of the number of cores available within the system. Furthermore, the order in which tasks are assigned to cores is still based on task utilization. We can do better. When tasks contend for cache regions, analysis of the cache conflict graph yields superior, conflict-guided allocations. Such analysis considers tasks in a conflict-conscious order that ensures they can co-exist with each other for a given number of cores. To this end, we adapted a graph coloring approach by Chaitin [5] that is widely used in register allocation, which is based on the following theorem:

**Chaitin's Theorem.** *Let G be a graph and v ∈ V(G) such that deg(v) < k, where deg(v) denotes the number of edges of vertex v. A graph G is k-colorable if and only if G - v is k-colorable.*

This theorem provides the bases for graph decomposition by repeatedly deleting vertices with degree less than *k* until either the graph is empty or only vertices with degree greater than or equal to *k* are left. In the latter case, the graph cannot be colored. However, by removing a task from a conflict graph using some heuristic, a new coloring attempt can be made for the remaining of the graph. Figure 3 shows how Chaitin's theorem can be used in practice. In this example, the conflict graph is the same as in the FFD example in Figure 4. This new example shows how Chaitin's approach allocates the set of nodes to two buckets/colors. At first, the algorithm fills up a stack removing one node at a time. A node is a viable candidate for being pushed onto the stack if and only if the degree is less than 2. When a node is removed, it reduces the degree of its neighbor in the remainder of the graph. Since all nodes can be pushed onto the stack, the graph is two-colorable (cf. Chaitin's theorem). During the following steps, nodes are popped off the stack and associated with a color/bucket. In our example, Chaitin's algorithm successfully allocates nodes to two buckets. In contrast, three buckets were required by the

**Input**: $M$ : Set of Tasks, $NumOfColors$ : Number of Cores $\times$ Number of locked ways per cache, $M_{conf}$ : conflict Matrix
**Output**: $colorList$ , $spilledList$, $rejectedTaskList$

```
1  colorStack := empty;
2  spilledList := empty;
3  colorList := empty;
4  while M is not empty do
5      M.sort(increasing degree_conflicts);
6      t := M.front();
7      if t.degree < NumOfColors then
8          push t onto colorStack ;
9          remove t from M and M_conf ;
       end
10     else
11         t := task with minimum (u_unlocked/degree) ;
12         push t onto spilledList ;
13         remove t from M and M_conf ;
       end
   end
14 aveCoreUtil = colorStack.u / NumOfColors;
15 while colorStack is not empty do
16     t := Pop colorStack ;
17     repopulate M_conf ;
18     while curColor < NumOfColors do
19         if None of the neighbors has this color then
20             curCore := curColor mod number Of Cores ;
21             if curCore.u < aveCoreUtil and curCore.u + t.u ≤ 1 then
22                 t.color := curColor ;
23                 colorList[curColor] := t ;
24                 Add t.u to curCore.u ;
               end
           end
       end
25     if t.color is not a valid Color then
26         push t onto rejectedTaskList ;
       end
   end
```

**Algorithm 2:** Task Coloring Algorithm

FFD algorithm.

Algorithm 2 shows the task coloring mechanism, which is responsible for finding non-conflicting tasks that can be grouped together in a given number of colors. The number of colors is equal to the number of locked cache ways that can be filled within a given number of cores. Lines 4-13 fill up two data-structures, $colorStack$ and $spilledList$. Every iteration of this loop finds a task that can be placed on either of these stacks. Line 5 sorts the tasks in increasing order of the number of tasks it conflicts with in $M$. A task with minimum degree is pushed onto $colorStack$ if and only if its degree is less than $NumOfColors$. Otherwise, the algorithm finds a task using a heuristic that focuses on minimizing the metric $u_{locked}/degree$ as shown in line 11. The objective of this heuristic is to decrease the conflict degrees of as many tasks as possible and, at the same time, to pick a task that causes the minimum increase in the system utilization while remaining unlocked ($u_{unlocked}$). This task is then added to the $spilledList$. While removing the tasks from $M$, we decrease the conflict $degree$ of neighbors. Once all tasks have been distributed among either of the stacks, lines 14-26 put the tasks in $colorStack$ into different colorLists. Assigning a task from $colorStack$ to a $colorList$ is equivalent to allocating the task to a core as each color corresponds to a lockable cache way. The $colorList$s are associated with cores in a round robin manner, i.e., if the number of lockable cache ways per task is equal to two and the number of cores is three, then there are a total of six $colorList$s. The first, second and third $colorList$s are associated with the first cache way on cores one, two and three, respectively. The fourth, fifth and sixth $colorList$s are associated with the second cache way on cores one, two and three. Lines 16-17 pop a task from the $colorStack$ and re-populate the conflict edges in the graph with the tasks that have already been colored. The algorithm then loops through all the colors until it finds a color that has not been allocated to any of its neighbors in the graph. Line 20 picks the core associated with that color. For a task to be assigned a color, the task has to pass the EDF schedulability test. Furthermore, the current utilization of the core has to be less than $aveCoreUtil$, where $aveCoreUtil$ has been computed at line 14. These conditions prevent $colorList$s from becoming unbalanced. Chaitin's algorithm in its purest form is (i) unaware of the tasks in the $spilledList$ and (ii) may deliver an unbalanced $colorList$. E.g., if none of the tasks are conflicting then all tasks can be given the same color. Conditions at line 24 allow the tasks to be evenly distributed across cores. If either of the conditions fail, then the algorithm moves on to the next color until all the colors have been tried. If a task cannot be assigned a valid color, it is moved to $rejectedTaskList$.

The task coloring stage outputs partially filled cores and a list of tasks in $rejectedTaskList$ and $spilledStack$. These are subsequently used by the second part of the allocation shown in Algorithm 3. Algorithm 3 first tries to allocate tasks from the $rejectedTaskList$. It sorts the tasks of $rejectedTaskList$ in decreasing order of their $u_{locked}$. Each iteration of the loop starting at line 2 then picks a task in order and tries to allocate it in FFD fashion on $N_{procs}$. If a task cannot be allocated to a core, it is moved to the $spilledList$. Once the $rejectedTaskList$ is empty, all the tasks in $spilledList$ are allocated in FFD manner while considering each task as unlocked. If all the tasks in $spilledList$ are allocated, the task set is deemed to be schedulable on a given number of $N_{procs}$ cores. Otherwise, $N_{procs}$ is incremented by the caller of CoFFD. This process repeats until a schedule has been found.

Figure 5 depicts a step-by-step working example:

(a) Tasks are grouped in a conflict graph. Our example has five tasks with $u_{locked}$ utilizations of 0.5, 0.3, 0.4, 0.2

**Input**: $rejectedTaskList$, $Assoc$ : Number of locked ways per cache, $M_{conf}$ : conflict Matrix,$N_{procs}$ : number of cores

```
1  rejectTaskList.sort(decreasing u_locked);
2  foreach rejectTaskList i do
3      N_procs.sort(decreasing u); Success = false;
5      foreach N_procs j do
6          foreach Assoc k do
7              if IsAllocatable(j,i,Assoc,M_conf) ≠ −1
               then
8                  allocate task i to core j in kth
                   associativity;
9                  j.u += i.u_locked;
10                 Success = true;
11                 goto ;
               end
           end
       end
12     if Success==false then
13         put task i on spilledList ;
       end
   end
14 spilledList.sort(decreasing u_unlocked);
15 foreach SpilledList i do
16     Success = false;
17     foreach N_procs j do
18         if i.u_unlocked < 1 - j.u then
19             allocate task i to core j;
20             j.u += i.u_unlocked;
21             Success = true;
22             break;
           end
       end
23     if Success==false then
24         return Failed Allocation;
       end
   end
25 return Successful Allocation;
```

**Algorithm 3:** Colored First Fit Decreasing (CoFFD)— Uncolored Lists

and 0.2. Each task conflicts with its neighboring task. Therefore, tasks form a chain of conflicts in the graph.

(b) Our graph coloring algorithm is applied to split the tasks in $ColorLists$. The task set is split into two colors alternating between adjacent tasks in the same $colorList$.

(c) We assume a multi-core system with single-way locking in the L1 cache. Since the aggregate utilization is 1.6, $N_{procs}$ is initialized with the ceiling of system utilization, which is 2. The tasks in each $colorList$ are sorted in decreasing order of $u_{locked}$. The cores are filled in a round-robin fashion. The green $colorList$ fits within core zero. Tasks in the red $colorList$ are allocated to core one. Tasks with higher utilization (0.5 and 0.4) are allocated to core one while the task with utilization 0.2 is moved to

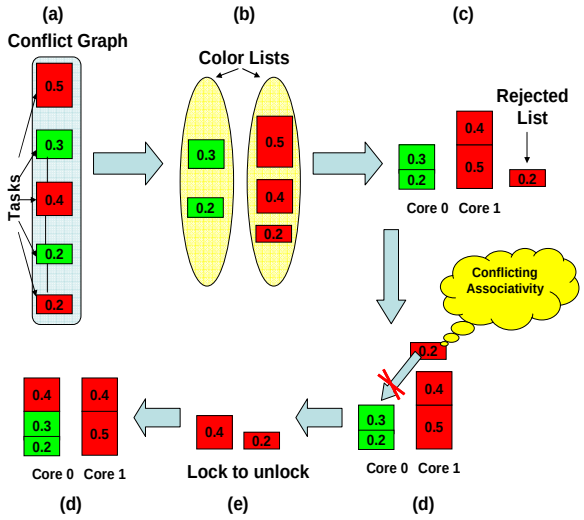


Fig. 5. Task Coloring in Operation

the $rejectedTaskList$ as it exceeds the utilization bound of 1.

(d) The algorithm now tries to allocate the task from $rejectTaskList$ to core zero. It fails due to task conflicts with an already allocated task and due to the availability of only one cache way for locking.

(e) At this stage, the task is moved to the $spilledList$. The task's utilization is increased to $u_{unlocked}$. This changes its utilization from 0.2 to 0.4.

(f) The task is allocated on core 0 with this inflated utilization because such allocation does not violate the utilization bound on core 0.

### 4.3. Optimized Region Resizing for Multi-cores

So far, we have assumed that conflicting tasks can only share a resource either by locking all specified regions or keeping all of them unlocked. This is useful when locked regions should remain transparent to the programmer. We can improve on our results if programmers can accurately estimate the upper bound on the number of references to each locked cache line (e.g., based on upper loop bounds). This requires the specification of the number of references ($N_{refs}$) for each locked cache line in $list_{locked\_set}$. We can then compute the reference frequency, $R_f$, of a locked cache line for task t as $R_f = \frac{N_{refs}}{Period_t}$

When the allocation of a task with $WCET_{locked}$ has failed, we need not inflate the WCET of the task directly from $WCET_{locked}$ to $WCET_{unlocked}$. Instead, we can resolve conflicts at a much finer granularity. If a task C has a conflict with another task A at set $m$, and if $R_f$ for set $m$ of task C is higher than $R_f$ for set $m$ of task A, then task C will retain its locked line while task A will lose one. If multiple cache ways are lockable, the locked cache line with the minimum $R_f$ is replaced. This increases the utilization of the task with the newly locked line.

**Optimization-induced changes to task allocation algorithms**: Since a task will lock multiple cache lines, allocation of a task to a core may affect different tasks on different cache sets. Hence, the schedulability test has to use the temporary WCETs of all the affected tasks before making permanent changes. The $IsAllocatable$ procedure performs a locked cache analysis and delivers the temporary WCETs along with a list of cache resizing specifications if the schedulability test succeeds. In case the test fails, the list of updates is rejected and no permanent changes are made to the WCETs. In Algorithm 2, the heuristic for selecting spilled tasks will change since partial locking of cache lines affects multiple tasks instead of dilating the WCET of just one. Thus, we spill the task whose $Number\_of\_conflicting\_cache\_lines/degree$) is minimal.

We can use the algorithms presented above in several ways. If tasks can meet their deadlines only under locking with $WCET_{locked}$, then these algorithms will allocate them with $WCET_{locked}$. If $WCET_{locked}$ and $WCET_{unlocked}$ are provided, then both fully locked and fully unlocked scenarios can be assessed by the algorithms. Dealing with execution times at coarser levels seems more attractive to the developers. This allows them to select lockable lines with rough estimate of the access patterns. Also, it may not be possible to deduce an accurate number of references or the estimates can be highly pessimistic, especially when data regions are being accessed sparsely. Conversely, if data regions are being frequently referenced and references are uniformly dense around locked regions, then the region resizing can be used in conjunction with CoFFD and CoFFD.

## 5. Task-set Generation

Due to the unavailability of a full blown real-time application for massive multi-core architectures, we decided to utilize synthetic task sets in our experiments. This also allows us to test corner cases of our algorithms. Table 2 shows the architectural and task-set parameters of our experimental framework.

| Parameter | Value |
|---|---|
| Processor Model | in-order |
| Cache Line Size | 32B |
| L1 Cache Size/Associativity | 8KB/2-way |
| Lockable associativity | 1/2 |
| L1 Access latency | 1 cycle |
| L2 Access Latency | 10 cycles |
| External Memory Latency | 100 cycles |
| Max. sets locked by a task | 114/128 |
| Min. sets locked by a task | 8/128 |
| Max. size of locked region | 57 sets |
| Min. size of locked region | 8 sets |
| Max. size of task-sets | 42 |
| total tasks generated | 126 |
| Max. locked regions by a task | 4 |
| Min. locked regions by a task | 1 |

TABLE 2. System Parameters

We generated the synthetic task-set values (period, locked execution time and unlocked execution time) as follows:

1) Task-sets with varying number of locked sets were generated. Tasks could have anywhere from 1 to 4 locked regions. Each locked region is given a random number of references. Every cache line is subjected to a uniform number of references to model spatial locality effects.
2) The total number of references were derived by aggregating the number of references incurred within the locked regions of the task. Since the programmer will be locking the regions in L1 (highest utilization benefit), we assume that these locked lines consume 80% of the total data loads. Out of the remaining 20%, we assume 18% are hits in the L2 cache and 2% are references to sensory data that goes off chip. We also assume that every 5th instruction is a load. This lets us infer number of instruction fetches that incur L1 cache hits (see Section 3). These assumptions allow us to derive a $WCET_{locked}$ for a task.
3) To derive the $WCET_{unlocked}$, we assume unlocked regions to hit in L2 cache. If two locked regions are accessed by two different paths, then the increase in WCET is due to just one region (the one that dominates the references), not both. Thus, we randomly select tasks to accommodate such behavior. This also results in varied increases in execution time between $WCET_{locked}$ and $WCET_{unlocked}$ across tasks.
4) Next, we assign periods to each task $i$ to group them into different utilization categories: high utilization $(0.55 > u_i > 0.40)$, medium utilization $(0.40 > u_i > 0.25)$, and low utilization $(0.25 > u_i > 0.15)$.

## 6. Evaluation

This section presents results of the conducted experiments. We present our experimental results for a system that supports single locked cache ways. Such a scheme is also applicable when considering horizontal cache partitioning, where all the lockable ways in each set are dedicated to a task.

TABLE 3. Allocated Cores for CoFFD & GFFD: All Tasks Locked

| Number of Tasks | High Util. | | Med. Util. | | low Util. | |
|---|---|---|---|---|---|---|
| | GFFD | CoFFD | GFFD | CoFFD | GFFD | CoFFD |
| 4 | **3** | **3** | 3 | **2** | 3 | **2** |
| 8 | 6 | **5** | 5 | **4** | **4** | **4** |
| 12 | 9 | **8** | 6 | **5** | **5** | **5** |
| 16 | 11 | **10** | 9 | **8** | **8** | **8** |
| 20 | **13** | **13** | 12 | **11** | 12 | **11** |
| 24 | 16 | **15** | 16 | **15** | 16 | **15** |
| 28 | 20 | **19** | 20 | **19** | 20 | **19** |
| 32 | 22 | **20** | 22 | **21** | 22 | **21** |
| 36 | 24 | **21** | 24 | **22** | 23 | **22** |
| 42 | 27 | **25** | 25 | **24** | 24 | **23** |

**Allocations while retaining of locked state**: Table 3 depicts the results of our algorithms when tasks are allocated in locked

state, i.e., with an execution time of $WCET_{locked}$. The first column shows the number of tasks in the task-set. The third and fourth columns show the number of cores allocated by GFFD and CoFFD, respectively, when a task-set is composed of high utilization tasks only. The fifth and sixth columns represent the same for medium utilization tasks, and the sixth and seventh columns for lower utilization tasks. Lower core allocations are depicted in bold font. In all cases, CoFFD results in fewer cores allocated than GFFD, especially as the number of tasks increases. As more tasks are added to the system, the conflict graph becomes denser. CoFFD will then avoids strategically conflicts under due to its coloring scheme while the greedy scheme results in a less conflict-conscious allocation.

TABLE 4. CoFFD vs GFFD: Selected Tasks Unlocked

| **Number of Tasks** | GFFD | CoFFD | GFFD Util. | CoFFD Util. | Util. decreased by CoFFD |
|---|---|---|---|---|---|
| 4 | **2** | **2** | 1.48 | 0.88 | **40.54 %** |
| 8 | **3** | **3** | 2.05 | 2.027 | **0.88 %** |
| 12 | 5 | **4** | 3.77 | 3.06 | **18.83 %** |
| 16 | 7 | **6** | 5.07 | 4.13 | **18.54 %** |
| 20 | 9 | **8** | 7.33 | 5.86 | **19.64 %** |
| 24 | 11 | **10** | 8.6 | 7.04 | **18.13 %** |
| 28 | 12 | **11** | 10.2 | 8.65 | **15.19 %** |
| 32 | 14 | **12** | 11.57 | 9.7 | **16.16 %** |
| 36 | 15 | **15** | 12.67 | 10.27 | **18.94 %** |
| 42 | 17 | **17** | 14.04 | 11.87 | **20.37 %** |

**Allocations with all or none**: This experiment allows allocation of tasks either with locking of all regions or while leaving all of them unlocked. After a locked allocation with $WCET_{locked}$ is attempted, algorithms can fall back to an unlocked allocation with $WCET_{unlocked}$ for a given task in case conflicts have prevented the allocation on a given core. Table 3 depicts the results with best results in bold face. The first column shows the number of tasks in the task-set. The second and the third columns show the number of cores allocated by GFFD and CoFFD, respectively. Sets with higher/medium utilization tasks result in similar allocations. This is because it is difficult for the higher utilization tasks to be allocated under the inflated execution budget of $WCET_{unlocked}$. However, tasks with lower utilizations can be allocate tasks with $WCET_{unlocked}$. The fourth and the fifth columns depict the system utilization delivered under the allocations of the algorithms. The last column shows the decrease in system utilization achieved by CoFFD over GFFD. The results indicate that CoFFD beats GFFD not only in terms of allocating fewer cores but has in improving system utilization by over 18% for task-sets with large numbers of tasks. This is because GFFD inflates the execution budget of task that cannot be allocated to cores under locking. In addition, conflict analysis prior to allocation allows the algorithm to apply heuristics to reduce the number of tasks that remain unlocked. The results of CoFFD are due to a combined heuristics for selecting spilled tasks in CoFFD. Heuristic 1 selects the task with the least $\frac{WCET_{unlocked}}{degreeofConflicts^2}$ value, which emphasizes the task's degree. This prevents the number of cores to be increased when non-conflict placements are still feasible. Algorithmically, CoFFD avoids spills of tasks onto the stack (see Algorithm 3). Heuristic 2 selects the task with the least $WCET_{unlocked}$ value. Of the two heuristics, CoFFD selects the one that results in the allocation of fewer cores. For example, most task sets in Table 4 resulted in the allocation of fewer cores under heuristic 1, but the last task set would have resulted in the allocation of 18 cores whereas heuristic 2 reduced this allocation to 17. This behavior was also observed while allocating tasks with locked region resizing (see below).

TABLE 5. Region Resizing

| **Number of Tasks** | GFFD w/ Partial Locking | CoFFD w/ Partial Locking | GFFD w/ locks only | CoFFD w/ locks only |
|---|---|---|---|---|
| 4 | **2** | **2** | 3 | **2** |
| 8 | **3** | **3** | **4** | **4** |
| 12 | **4** | **4** | **5** | **5** |
| 16 | **6** | **6** | **8** | **8** |
| 20 | **7** | **7** | 12 | **11** |
| 24 | **8** | **8** | 16 | **15** |
| 28 | **10** | **10** | 20 | **19** |
| 32 | **10** | **10** | 22 | **21** |
| 36 | **12** | **12** | 23 | **22** |
| 42 | **13** | **13** | 24 | **23** |

**Region Resized Locking**: The next experiment assessed the optimization of resizing locked regions for conflicted tasks. We observed that sets with high utilization tasks result in dilation of WCET when locking fails, which reduces their chances of being allocated. In Table 5, we show the results for task-sets with low utilization tasks as they benefited the most from region resizing. The first column shows the number of tasks in the task-sets. The second and the third columns show the number of cores allocated when partial locking is used by GFFD and CoFFD, respectively. The fourth and fifth columns show the number of allocated cores when tasks are not allowed to unlock any of their regions. The results indicate that for higher number of tasks, partial locking after resizing reduces the number of required cores by 50%. It is interesting to note that the greedy algorithm performed as well as CoFFD with combined heuristics 1 and 2. This is due to the fine-grained arbitration of conflict regions under resizing. For task-sets with medium utilization tasks, CoFFD and GFFD allocate a similar number of cores for all task-sets. Yet, CoFFD results in 1%-14% reduced system utilization.

## 7. Conclusions

The use of multi-core architectures is not yet prevalent in real-time systems since guaranteeing predictability of hard real-time tasks on such architectures remains a challenge. Cache locking is a technique that is commonly employed to improve the predictability of real-time task execution. Multiple tasks may choose to lock conflicting regions in the cache. While multi-core architectures can conceptually support such scenarios by allocating conflicting tasks onto different cores,

current static task partitioning techniques for hard real-time tasks do not take such conflicts into account.

This paper proposes a two algorithms for task allocation in a multi-core environment where tasks are allowed to lock cache lines in a specified subset of cache ways for each core's private L1 cache. The first algorithm, GFFD is an enhanced versions of the First Fit Decreasing (FFD) algorithm. The second, CoFFD, is based on a graph coloring method. Experimental results indicate that CoFFD consistently performs better than the GFFD for lower number of cores and lower system utilization.

We also propose a mechanism that allows locked regions to be resized. This scheme is applicable when the programmer can accurately provide the number of references to a locked cache line, yet does not want to be concerned with fine-grained locking decisions. The two algorithms were further adapted to use task and reference information to choose whether to retain a line in locked or unlock state for conflicting regions. With such partial locking, the number of cores in some cases is reduced by almost 50% with an increase in system utilization of 10%. Overall, this work is unique in considering the challenges of future multi-core architectures for real-time systems and provides key insights into task partitioning with locked caches for architectures with private caches.

## References


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

[2] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 179–190, Apr. 2006.

[3] 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.

[4] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Euromicro Conference on Real-Time Systems*, pages 209–308, July 2008.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 98–105, 1982.

[6] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM.

[7] D. Choffnes, M. Astley, and M. J. Ward. Migration policies for multi-core fair-share scheduling. *ACM SIGOPS Operating Systems Review*, 42:92–93, 2008.

[8] N. Eisley, L.-S. Peh, and L. Shang. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *International conference on Parallel architectures and compilation techniques*, pages 197–207, 2008.

[9] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 245–254, New York, NY, USA, 2009. ACM.

[10] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 68–77, Washington D.C., USA, Dec. 2009.

[11] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *In ACM/IEEE conference on Supercomputing*, pages 1–11, Nov. 2007.

[12] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. McElderry, and S. Hahn. Operating system support for shared-isa asymmetric multi-core architectures. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 19–26, June 2008.

[13] J. Ouyang and Y. Xie. Loft: A high performance network-on-chip providing quality-of-service support. *Microarchitecture, IEEE/ACM International Symposium on*, 0:409–420, 2010.

[14] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*, pages 57–68, 2009.

[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 *Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[18] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

[19] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *In IEEE Real-Time Systems Symposium*, pages 154–165, 2003.