Static Task Partitioning for Locked Caches in Multi-Core Real-Time Systems[*]

Abhik Sarkar, Frank Mueller, North Carolina State University
and Harini Ramaprasad, Southern Illinois University Carbondale

Growing processing demand on multi-tasking real-time systems can be met by employing scalable multi-core architectures. For such environments, locking cache lines for hard real-time systems ensures timing predictability of data references and may lower worst-case execution time. This work studies the benefits of cache locking on massive multicore architectures with private caches in the context of hard real-time systems. In shared cache architectures, the cache is a single resource shared among *all* the tasks. However, in scalable cache architectures with private caches, conflicts exist only among the tasks scheduled on one core. This calls for a cache-aware allocation of tasks onto cores.

The objective of this work is to increase the predictability of memory accesses resolved by caches while reducing the number of cores for a given task set. This allows designers to reduce the footprint of their subsystem of real-time tasks and thereby cost, either by choosing a product with fewer cores as a target or to allow more subsystems to be co-located on a given fixed number of cores.

Our work proposes a novel variant of the cache-unaware First Fit Decreasing (FFD) algorithm called Naive locked First Fit Decreasing (NFFD) policy. We propose two cache-aware static scheduling schemes: (1) Greedy First Fit Decreasing (GFFD) and (2) Colored First Fit Decreasing (CoFFD) for task sets where tasks do not have intra-task conflicts among locked regions (Scenario A). NFFD is capable of scheduling high utilization task sets that FFD cannot schedule. Experiments also show that CoFFD consistently outperforms GFFD resulting in lower number of cores and lower system utilization. CoFFD reduces the number of core requirements by 30% to 60% compared to NFFD.

For a more generic case where tasks have intra-task conflicts, we split the task partitioning between two phases: Task selection and task allocation (Scenario B). Instead of resolving conflicts at a global level, these algorithms resolve conflicts among regions while allocating a task onto a core and unlocking at region level instead of task level. We show that a combination of dynamic ordering (task selection) with Chaitin's Coloring (task allocation) scheme reduces the number of cores required by up to 22% over a basic scheme (in a combination of monotone ordering and regional FFD). Regional unlocking allows this scheme to outperform CoFFD for medium utilization task sets from Scenario A. However, CoFFD performs better than any other scheme for high utilization task sets from Scenario A. Overall, this work is unique in considering the challenges of future multicore architectures for real-time systems and provides key insights into task partitioning and cache-locking mechanisms for architectures with private caches.

Categories and Subject Descriptors: B.4.2 [**Memory Structures**]: Design Styles,cache memories; D.4.7 [**Operating Systems**]: Organization and Design,real-time systems and embedded systems; D.4.1 [**Operating Systems**]: Process Management,scheduling

General Terms: Design, Experimentation

Additional Key Words and Phrases: Real-Time Systems, Multi-Core Architectures,Timing Analysis

1. INTRODUCTION

Multicore architectures have become prevalent in embedded system design. This is evident from the variety of multicore processors available today, such as the 4-core MPCores and Cortex processors from ARM, the 8-core P4080 PowerPC from Freescale and large multicores like the 64-core TilePro64 from Tilera, Adapteva's 64-core Par-

[*]This work was supported in part by NSF grants 0905181, 0905212 and 1239246. This journal version is an extension to prior work [Sarkar et al. 2012]. This journal introduces a set of novel algorithms for task-sets that have intra-task conflicts (Scenario B). This has been listed 4th in our summary of contributions in Section 1. It adds two whole new sections: (1) Section 4,(2) Section 6.2. In Section 5, we explain the inclusion of Tgff [Dick et al. 1998] to generate a large set of task-sets to introduce a wide range of intra- and inter-task conflicts. In Section 3, we illustrate the inefficiency of CoFFD and GFFD in dealing with Scenario B. Authors' addresses: Abhik Sarkar and Frank Mueller, Dept. of Computer Science, North Carolina State University; Harini Ramaprasad, Dept. of Computer Science, Southern Illinois University Carbondale.

allella, Intel's 48-core SCC and Kalray's 256-core MPPA [ARM ; Freescale ; Tilera 2009; Adapteva ; Howard et al. 2010; de Dinechin et al. 2013], 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 multicore caches have been a significant contributing factor to this skepticism.

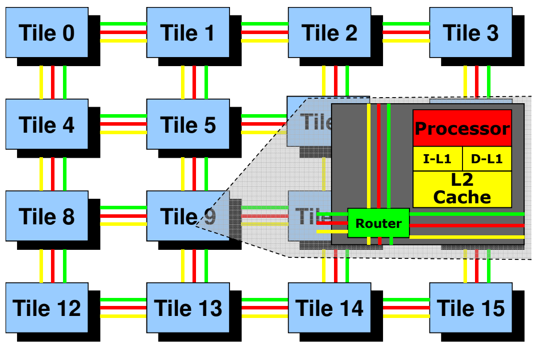

Fig. 1. Tile-based Architecture

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 [Chattopadhyay et al. 2010; Guan et al. 2009]. Such schemes become inapplicable to scalable multicores, such as shown in Figure 1. These architectures use private L1+L2 caches. Any task allocation algorithm on such architectures 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 with high associativity (16-32 ways) 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 multicore platforms.

In general, cache locking techniques provide predictability to a task's cache access behavior [Busquets-Matraix 1996; Puaut and Decotigny 2002; Puaut 2006; Puaut and Pais 2007; Vera et al. 2003; 2007; Suhendra and Mitra 2008; Liu et al. 2010; Liu et al. 2009; Plazar et al. 2012]. Cache locking can be realized at various granularities. Studies on uniprocessor cache locking have assumed the entire L1 cache to be locked [Puaut and Decotigny 2002; Puaut and Pais 2007]. Another study on cache locking for shared caches has assumed locking individual cache lines [Suhendra and Mitra 2008]. Locked caches on uniprocessors 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 multicore 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 a disjoint set of cores.

Prior literature on uniprocessor locking techniques focuses on filling a single cache way while reducing the overall utilization of a core. When cache locking is localized, we refer to a *region* of code, where upon entry and exit lines are locked and unlocked, respectively. The objective of allocating tasks on scalable multicores has to be balanced between the following objectives:

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

Reduction of the system utilization can be achieved by placing all tasks with conflicting locked cache regions on different cores on scalable architectures. 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, a good solution needs to consider the trade-off between these objectives.

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.

The objective of this work is to increase the predictability of memory accesses resolved by caches while reducing the number of cores for a given task set. This allows designers to reduce the footprint of their subsystem of real-time tasks and thereby cost, either by choosing a product with fewer cores as a target or by allowing more subsystems to be co-located on a given fixed number of cores.

In this work, we split the problem into two scenarios.

(1) Scenario A presents the problem with task sets of locked regions that can fit within a cache way. These task sets do not have any intra-task cache conflicts by design. It is further shown why naive solutions are inadequate. One of the most commonly used partitioning algorithms is the First Fit Decreasing (FFD) algorithm. First, we extend this algorithm with an approach called Naive Locked FFD (NFFD). We call these algorithms "cache unaware" as they avoid any form of analysis on locked cache regions. Then, we develop and evaluate two cache-aware partitioning algorithms: (1) Greedy First Fit Decreasing (GFFD), and (2) Colored First Fit Decreasing (CoFFD). GFFD tries to allocate tasks onto a minimum number of cores [Burchard et al. 1995]. CoFFD, a more sophisticated scheme, exhibits a novel approach based on graph coloring that delivers task partitioning.
(2) Scenario B looks into a more generic case: Tasks can have locked regions that cause intra-task conflicts and thus require multiple cache ways to avoid such conflicts. This renders the algorithms mentioned above inadequate. We tackle this problem, detailed in Section 4, by splitting task partitioning into two phases: Task selection and task allocation. Task selection algorithms pick a task in some order and task allocation algorithms try to resolve regional conflicts at individual cores while allocating tasks onto them. We present two task allocation mechanisms, namely Regional First Fit Decreasing (RFFD) and Chaitin's Coloring (CC). We further present two task selection mechanisms, namely Monotonic (Mono) and Dynamic (Dyn).

To establish the significance these algorithms can have on allocation, we depict them in Table 1. Here, we show a comparison of the number of allocated cores for different **Scenario A task sets** of 32 tasks using FFD, NFFD, GFFD and CoFFD on an archi-

Table I. Locking and Conflict Analysis for 32 Tasks

Task Set Type	Number of Cores Required			
	FFD	NFFD	GFFD	CoFFD
High util.	Failed	32	22	20
Med. util.	31	31	21	20
Low. Util.	23	22	14	12

tecture that uses the same system parameters for our experimental results (shown in Section 5 as Table II). We consider two utilizations for each task: one with locking for all regions specified by the developer (u_{locked}) and another without locking any of those regions ($u_{unlocked}$). A task is termed to be of high, medium and low utilization when $(0.55 > u_{locked} \geq 0.40)$, $(0.40 > u_{locked} \geq 0.25)$ and $(0.25 > u_{locked} \geq 0.10)$, respectively. The first column depicts the number of tasks in the task sets. The remaining columns show the number of cores consumed by the task set under FFD, NFFD, GFFD and CoFFD, respectively. We observe that CoFFD consistently results in allocating fewer cores than GFFD. Task sets composed of high utilization tasks allocate fewer cores under CoFFD with at most 3% higher system utilization than GFFD. For low utilization task sets, CoFFD allocates fewer cores and lowers system utilization by up to 40% over GFFD. This has been the basis for using efficient coloring algorithms as discussed next.

For task sets from Scenario B, the combination of Mono+CC outperforms Mono+RFFD for highly conflicted task sets. This shows the effectiveness of using the coloring mechanism at individual cores. However, for low contention task sets, it does not seem as effective. One key insight is that a global ordering among tasks is as important as ordering of regions among co-located tasks. This is substantiated by Dyn+CC as it consistently performs best among all scenario B algorithms. For high contention task sets, Dyn+CC results in reductions of up to 22% in the number of allocated cores, i.e., it allocates 21 cores as opposed to 27 cores (Mono+RFFD). Even for low contention task sets, it is able reduce cores by up to 17%. Since Dyn+CC tackles a more generic problem, it is also applicable to task sets from Scenario A. While comparing the CoFFD with Dyn+CC, one key insight is that CoFFD performs better that Dyn+CC for high utilization tasks. That is because unlocking a region or a task does not allow multiple tasks to be scheduled together. Thus, the global ordering achieved by CoFFD is better. However, CoFFD is unable to unlock entire tasks for medium utilization tasks, yet unlocking at regional granularity is feasible. CoFFD is consequently outperformed by Dyn+CC. Overall, Dyn+CC delivers the best allocation for Scenario B task sets. For Scenario A task sets, both CoFFD and Dyn+CC are equally competitive.

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

(1) We implement two task partitioning algorithms for Scenario A type task sets: GFFD and CoFFD. These algorithms resolve the conflicts at task level by selectively locking or unlocking tasks.
(2) Our novel CoFFD algorithm (i) derives task allocations for a given number of cores resulting in a feasible schedule, (ii) enhances a coloring algorithm to deliver balanced allocation and (iii) reduces the number of cores relative to Greedy First Fit Decreasing (GFFD).
(3) For Scenario B, we propose a novel mechanism to resolve conflicts at the granularity of regions. We propose a Dynamic (Dyn) ordering mechanism that adapts to the changes in the regional conflict graph induced by the allocation of tasks to cores. Dynamic ordering consistently allows the allocation of tasks on fewer cores with both Task Allocation algorithms. Dyn+CC proves to be the best among all the combinations of Task Selection and Task Allocation Schemes.

Overall, our locking for multicores ensures that the number of cores for a given task set can be kept low while tight WCET bounds can be derived. This is accomplished via L1 cache locking to provide isolation between tasks so that inter-task analysis for WCET becomes less pessimistic. Without our approach, large task blocking terms have to be considered in schedulability analysis due loose bounds on inter-task conflicts and the number of preemption points, both of which are challenging problems adding to pessimism that may render task sets infeasible in terms of real-time schedulability. In contrast, our approach only requires intra-task WCET analysis, which can be much tighter and does not require to be complemented by inter-task blocking terms due to cache replacement. WCET analysis is easily adapted to consider fixed latencies for L1 locked memory regions to derive safe and much tighter WCET bounds.

2. ASSUMPTIONS, LIMITATIONS AND SYSTEM DESIGN

This work explores the challenge of utilizing many-core architectures in the domain of real-time system applications while providing predictable execution guarantees. The focus is upon allocating independent tasks on many-core architectures with distributed caches. This is driven by contemporary and emerging mesh multicore architectures with private caches per core. The approach holds for any upper-level hierarchy of private caches, but last-level caches that are shared are beyond the scope. The paper highlights the possibility of significantly reducing the core usage utilizing cache conflicts for task-to-core allocations. It assumes that tasks are independent, i.e., a generalization of the approach to dependent tasks. For dependent tasks, a partial blocking calculation based on inter-task conflicts has the potential for significantly less pessimism if the set of shared lines is reduced, an idea subject to future work.

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 multicore 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 [Ouyang and Xie 2010]. 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.

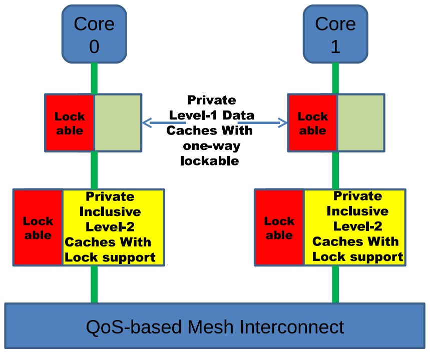
Core 0
Core 1
Lock able
Private Level-1 Data Caches With one-way lockable
Lock able
Lock able
Private Inclusive Level-2 Caches With Lock support
Lock able
Private Inclusive Level-2 Caches With Lock support
QoS-based Mesh Interconnect

Fig. 2. A Lock-based Architecture

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 [Hardy et al. 2009]). 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 has been considered in recent research [Paolieri et al. 2009]. Here, we analyze two scenarios

(1) A hard real-time task can only lock one cache line per set. All the locked regions of a task can fit within a direct mapped L1 cache. So, for a 8KB L1 cache with an associativity of two, a hard real-time task can lock up to 4KB of cache content. We call this Scenario A.
(2) A hard real-time task can lock multiple cache lines per set. Here, conflicting locked regions are able to occupy multiple cache ways. So, for a 8KB L1 cache with an associativity of two, a hard real-time task can lock all of the 8KB cache space or any subset at cache-line granularity. We call this Scenario B.

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 uniprocessor, 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.

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.

The paper contributes allocation schemes for different locking schemes, which are driven by contemporary architectural support for locking data in caches. While some architectures support locking at the granularity of an entire way of an associative cache, others support finer-grained line-based locking. These mechanisms are reflected in scenarios "A" and "B" in this work. Both are evaluated under an extensive experimental framework with task-level simulations that considers a multitude of task sets with different utilization characteristics. More fine-grained cycle-level simulations with different network-on-chip (NoC) interconnect topologies and memory contention are beyond the scope. In particular, a interesting future direction is to bounded memory access latencies on the NoC and at each memory controller of a chip, e.g., by partition the memory to reduce and analyze potential conflicts of references, coupled with the task-to-core allocation policies developed in this paper. In fact, one of the promising directions is to study task isolation techniques for shared resources so that analysis (such as in our work) can be constrained to the upper level of the memory hierarchy while lower levels are not impacted by inter-task dependencies. Software partitioning has been proposed for paging, caches and memory banks [Wolfe 1993; Mueller 1995; Liedke et al. 1997; Busquets-Matraix 1997; Puaut and Hardy 2007; Mancuso et al. 2013; Ward et al. 2013; Herter et al. 2011; Yuny et al. 2014], and hardware proposals exist as well [Akesson et al. 2007; Suhendra and Mitra 2008; Paolieri et al. 2009]. These mesh nicely with our proposed solutions.

3. TASK PARTITION ALGORITHMS: SCENARIO A

3.1. Cache-Unaware Schemes

Static task partitioning algorithms for multicore architectures have been widely studied. Most of these approaches consistently aim at minimizing the number of cores utilized [Burchard et al. 1995]. They use bin-packing schemes considering a single utilization value per task. These algorithms for distributed systems are cache unaware. In the following section, we present two cache-unaware schemes, namely FFD and NFFD.

3.1.1. First Fit Decreasing (FFD). FFD is a commonly used algorithm for allocating tasks on distributed cores. This implementation assumes that the tasks are unlocked, i.e., we consider all tasks with a utilization of $u_{unlocked}$ using $WCET_{unlocked}$. This algorithm takes task (i), already allocated set of cores N_{procs} and a flag that decides whether task to be allocated in a locked state or unlocked state if it adds a new core to N_{procs}. The FFD algorithm picks tasks in decreasing order of their $u_{unlocked}$ and allocates them using Algorithm 1. Line 1 sorts the cores in N_{proc} in decreasing order of core utilization. Lines 3-8 iterate over the cores until the task is allocated or until all cores have been considered and task could not be allocated. A task is allocated to a core if a core's utilization does not exceed 1 (utilization bound for EDF). If a task could not be allocated to any core in N_{procs}, lines 9-13 add a new core to N_{procs} and the task is allocated to it in an unlocked state.

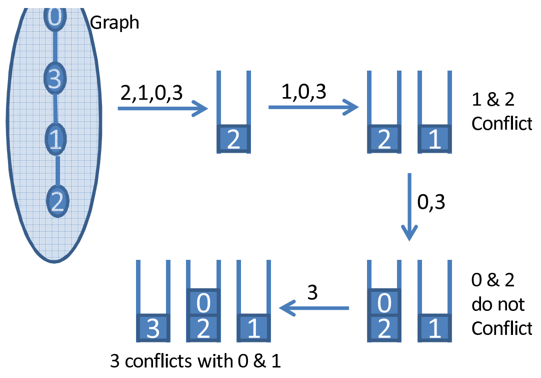

Fig. 3. Greedy First Fit Decreasing in Operation

3.1.2. Naive Locked FFD (NFFD). We extend FFD with a simple approach of using locked caches. Tasks are defined to be locked or unlocked prior to their allocation. Thus, all the tasks have a single WCET before allocation, which is $WCET_{locked}$ for a locked task or $WCET_{unlocked}$ otherwise. Bin packing has difficulties to co-locate multiple tasks with

ALGORITHM 1: FFD Task Allocation (baseFFD)

```
Input: $i$ : task, $N_{procs}$ : processors, $isLock$: boolean
Output: $N_{procs}$ number of processors
1  $N_{procs}$.sort(decreasing utilization) ;
2  foreach $N_{procs}$ $j$ do
3      if $Success$ = false then
4          if $i.u_{unlocked} \leq 1 - j.u$ then
5              allocate task $i$ to core $j$;
6              $j.u = j.u + i.u_{unlocked}$;
7              $Success$ := true ;
8              break;
           end
       end
   end
9  if $Success$ = false then
10     allocate $New_{proc}$;
11     $N_{procs}$ := $N_{procs} \cup New_{proc}$;
12     allocate task $i$ to $New_{proc}$;
13     if $isLock$ = true then
           $New_{proc}.u$ := $i.u_{locked}$;
       end
       else
           $New_{proc}.u$ := $i.u_{unlocked}$;
       end
   end
```

high utilization. Any task whose utilization is greater than 1 is deemed to be locked. Each of these locked tasks is allocated to a separate core as the algorithm is cache-unaware. The algorithm proceeds to allocate the set of unlocked tasks with an initial value of N_{procs}, the number of cores assigned to locked tasks.

3.2. Cache-Aware Task Partitioning

We next present two cache-aware mechanisms. Initially, our algorithms consider two values, $WCET_{locked}$ and $WCET_{unlocked}$. 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.

Definition 3.1 (*Conflict Matrix*). Given a set of N tasks, M_{conf} is an $N \times N$ matrix such that

$$M_{conf}(i, j) = \begin{cases} 0, & \text{if } cachelines(T_i) \cap cachelines(T_j) = \emptyset \\ 1, & \text{otherwise.} \end{cases}$$

3.2.1. Greedy First Fit Decreasing (GFFD). We first illustrate GFFD by example using a conflict graph. An undirected conflict graph of four nodes/vertices is depicted in Figure 3. 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 keeping the number of buckets low. The FFD algorithm arranges nodes in traversal order via heuristics (i.e., decreasing locked utilization) before allocating them. In this example, the algorithm establishes

ALGORITHM 2: Greedy First Fit Decreasing Heuristic (GFFD)

```
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 $k$:=$IsAllocatable(j,i,Assoc,M_{conf}) \neq -1$ then
9              if $i.u_{locked} \leq 1 - j.u$ then
10                 allocate task $i$ to core $j$ in $kth$ way;
11                 $j.u = j.u + i.u_{locked}$;
12                 $Success$ := true ;
13                 break ;
               end
           end
       end
14     if $Success$ = false then
15         $N_{procs}$ := $baseFFD$(i, $N_{procs}$,true);
       end
   end
```

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.

We developed a modified version of the FFD algorithm. We call this Greedy First Fit Decreasing (GFFD). Algorithm 2 presents the details of the algorithm. This algorithm takes a task set, the number of locked ways per cache and a conflict matrix M_{conf} as an input. The idea is to incrementally add cores to the schedule starting with an initial number of cores, N_{procs}, of 1. Lines 3-13 proceed to allocate tasks in FFD fashion using u_{locked}. Line 8 uses a procedure IsAllocatable() that 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. 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, line 15 uses Algorithm 1 to allocate the task. The value of $true$ for the third parameter to $baseFFD$ forces the task to be allocated in locked state when a new core is added to N_{procs}.

3.2.2. 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

ALGORITHM 3: Task Coloring Algorithm

```
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      $t$ := lowest degree task by linear search of $M$ ;
6      if $t.degree < NumOfColors$ then
7          push $t$ onto $colorStack$ ;
8          remove $t$ from $M$ and $M_{conf}$ ;
       end
9      else
10         $t$ := task with minimum ($u_{unlocked}/degree$) ;
11         push $t$ onto $spilledList$ ;
12         remove $t$ from $M$ and $M_{conf}$ ;
       end
   end
13 $aveCoreUtil$ = $\frac{colorStack.u}{NumOfColors}$;
14 while $colorStack$ is not empty do
15     $t$ := Pop $colorStack$ ;
16     repopulate $M_{conf}$ ;
17     $curColor$:=0;
18     for $curColor = 0 \to NumOfColors - 1$ do
19         if None of the neighbors has this color then
20             $curCore := curColor \mod$ number Of Cores ;
21             if $curCore.u < aveCoreUtil$ and $curCore.u + t.u \leq 1$ then
22                 $t$.color := $curColor$ ;
23                 $colorList$[curColor] := $t$ ;
24                 Add $t$.u to $curCore$.u ;
25                 break ;
               end
           end
       end
26     if $t$.color is not a valid Color then
27         push $t$ onto $rejectedTaskList$ ;
       end
   end
```

with each other for a given number of cores. To this end, we adapted a graph coloring approach by Chaitin [Chaitin 1982] that is widely used in register allocation, which is based on the following theorem:

CHAITIN'S THEOREM 1. *Let G be a graph and $v \in V(G)$ 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 basis 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 4 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 3. 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 FFD algorithm.

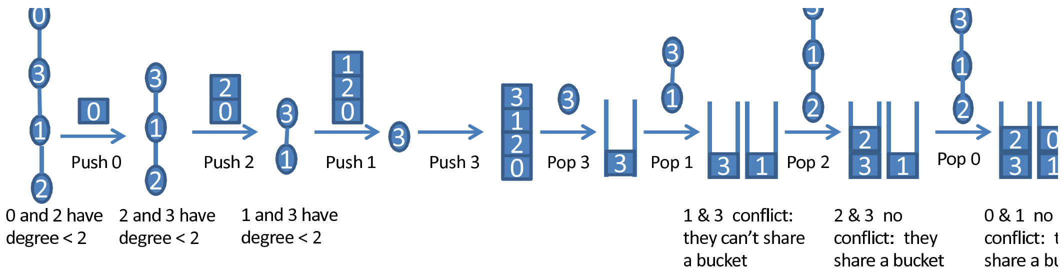

Fig. 4. Chaitin's Coloring in Operation with 2 Colors

Algorithm 3 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 searches through the list of unallocated tasks and finds the task with lowest degree. 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 a metric. For example, in algorithm 3 the metric $u_{locked}/degree$ is minimized at line 10. 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 13-27 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 15-16 pop a task from the $colorStack$ and repopulate 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 $aveCoreUtil$ is computed at line 14. These conditions prevent $colorList$s from becoming unbalanced. Chaitin's algorithm in its purest form is

— unaware of the tasks in the $spilledList$ and
— unable to deliver a balanced $colorList$.

E.g., if none of the tasks are conflicting then all tasks can be given the same color. Conditions at line 21 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 $rejectedTaskList$.

ALGORITHM 4: Colored First Fit Decreasing (CoFFD)— Uncolored Lists

```
Input: $rejectedTaskList$, $Assoc$ : Number of locked ways per cache, $M_{conf}$ : conflict
       Matrix,$N_{procs}$ : number of cores
Output: $Allocation\ status$
1  $rejectedTaskList$.sort(decreasing $u_{locked}$);
2  foreach $rejectedTaskList\ i$ do
3      $N_{procs}$.sort(decreasing $u$); $Success$ = false;
4      foreach $N_{procs}\ j$ do
5          foreach $Assoc\ k$ do
6              if $IsAllocatable(j,i,Assoc,M_{conf}) \neq -1$ then
7                  allocate task $i$ to core $j$ in $kth$ associativity;
8                  $j.u$ = $j.u$ + $i.u_{locked}$;
9                  $Success$ = true;
10                 goto line 11;
               end
           end
       end
11     if $Success$==$false$ then
12         put task $i$ on $spilledList$ ;
       end
   end
13 $spilledList$.sort(decreasing $u_{unlocked}$);
14 foreach $SpilledList\ i$ do
15     if $N_{procs} \neq baseFFD(i,N_{procs},false)$ then
16         return Failed Allocation;
       end
   end
17 return Successful Allocation;
```

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 4. Algorithm 4 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 using $baseFFD$. 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:

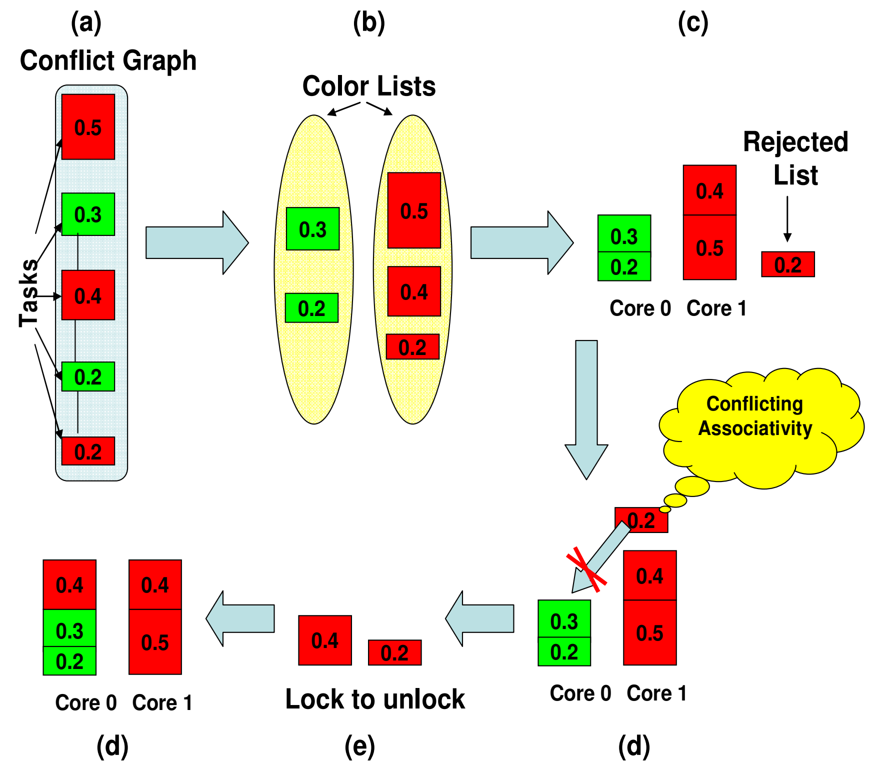

Fig. 5. Task Coloring in Operation

(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 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 $ColorList$s. The task set is split into two colors alternating between adjacent tasks in the same $colorList$.
(c) We assume a multicore 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 the $rejectedTaskList$ as it exceeds the utilization bound of 1.

(d) The algorithm now tries to allocate the task from $rejectedTaskList$ 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 $u_{unlocked}$ because the previous steps show that the task cannot be allocated on given cores without unlocking its locked regions. 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.

3.2.3. Algorithmic Complexity for Task Locking. Bin packing is known to be NP-hard. Any known optimal solution is exponential in complexity. Besides experimental evaluations, it is important to assess the complexity of sub-optimal, heuristic approaches to assess their scalable in terms of number of tasks and cores. In the following, the algorithmic complexity of GFFD and CoFFD are assessed.

For the purpose of complexity analysis, let the number of tasks be X and the number of cores be Y. Let t be the task to be allocated next.

Algorithmic Complexity of GFFD: The outer loop in algorithm 2 iterates over all tasks. The inner loop from 8-13 iterates over all cores. The function $IsAllocatable$ iterates over the task task conflict set, M_{conf}, bounded by the number of tasks, to detect if t conflicts with any of the tasks allocated to a core, i.e., $IsAllocatable$ has an algorithmic complexity of $O(X)$. Thus, the combined algorithmic complexity of GFFD is $O(YX^2))$.

Algorithmic Complexity of CoFFD: CoFFD consists of algorithms 3 and 4. The former algorithm colors the tasks while allocating them to cores. It has two loops. The first loop between lines 4 and 12 iterates over all tasks. The nested computations of linear search at line 5, reduction of number of conflicts for tasks conflicting with t at line 8 and 12, and linear search at line 10 are bounded by the number of tasks, i.e., they have an algorithmic complexity of $O(X)$ for a combined complexity of $O(X^2)$ for the first loop. The second loop between lines 14 and 27 iterates over all tasks while pushing them onto a stack. The nested loop within iterates over the set of colors, which is bounded by the number of cores, Y. The nested conditional at line 19 iterates over the set of neighboring nodes in the repopulated graph whose cardinality is bounded by the number of tasks, X. This implies an algorithmic complexity of $O(YX^2)$ for the second loop, which dominates the complexity of the first loop, i.e., is the overall algorithmic complexity of algorithm 3. The algorithmic complexity of algorithm 4, sequentially invoked next, is $O(YX^2)$ following the same argument as for GFFD since their algorithmic structure are equivalent in terms of loop iterators, i.e., the rejected task list is bounded by the number of cores. The loop iterating over the spilled list is bounded by the number of tasks but its complexity is dominated by the previous loop. Thus, the algorithmic complexity of CoFFD is $O(YX^2)$.

Thus, both the cache-aware algorithms deliver us the task partitioning with algorithmic complexity of $O(YX^2)$.

3.3. Discussion

Applicability to task sets from Scenario B: The algorithms discussed until now assume that the locked cache regions of a task fit within a single cache way. Task sets from Scenario B have intra-task conflicts within locked regions. Next, we present two techniques to apply the NFFD, GFFD and CoFFD on task sets from Scenario B:

(1) Lock only one region out of all conflicting regions: This could significantly increase the $WCET_{locked}$, which may render a task unschedulable.
(2) Retain all locked regions of a task that fit within the whole cache while treating all conflicting regions as one region that spans from the least indexed set to the maximum indexed set: Each core's cache can be treated as a direct mapped cache where a region is assumed to be spread across all the sets. This effectively partitions the cache horizontally. This solution does not have a notion of cache associativity and leads to inefficient task allocations.

As stated above, both techniques are highly inefficient. This is the motivation subsequently develop algorithms specifically for Scenario B. Such algorithms are presented next.

4. TASK PARTITION ALGORITHMS: SCENARIO B

The algorithms presented in this section allows tasks to have intra-task conflicts. Furthermore, a task can use multiple cache-ways as described in Section 2.

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 can be achieved through static analysis of tasks, i.e., by specifying the number of references (N_{refs}) for each locked cache region in $list_{locked\_set}$. We can then compute the reference frequency, R_f, of a locked region for task t as

$R_f = \frac{N_{refs}}{Period_t}$.

This becomes the basis for resolving conflicts at a finer granularity. However, the prior solutions do not provide us enough flexibility as conflicts are resolved at task level. The task partitioning algorithm for Scenario B is being split into two phases: Task Selection and Task Allocation.

Like GFFD, we use a greedy algorithm with a Task Selection mechanism to pick a task t for allocation from a list of tasks. Then, task t is being considered for allocation on every core until a core is found where it can be allocated along with the tasks that have already been allocated. The task allocation mechanism resolves conflicts between regions of task t and the tasks that have already been allocated onto the core. We use two heuristics for resolving regional conflicts, namely:

(1) Regional First Fit Decreasing (RFFD): RFFD, depicted in Algorithm 5, allocate a set of regions, RM, that belong to list of tasks, M, to a cache of associativity $Assoc$. In order to identify the conflicts among regions a conflict graph, M_{conf}, is required. Line 1 sorts the regions in decreasing order of their R_f. The loop beginning at line 2 iterates over each region to allocate them. Each iteration picks out a region from RM in line 4. Before every allocation attempt, a $Success$ flag is set to false. The nested loop starting at line 5 iterates over each cache way while considering to allocate region i to the a-th way. If region i does not have any conflicts with the regions already allocated in the a-th way, then IsAllocatable returns true; otherwise, it returns false. On success (true), region i gets allocated to a and the $Success$ flag is set to true before exiting the inner loop. Upon failure (false), the next associativity is considered. In case a region remains unallocated, the inner loop exits with $Success$ set to false. This causes the region to be placed in $unlocked\_regs$. Once all regions have either been allocated to cache ways or placed in $unlocked\_regs$, the

core's utilization $Util_{core}$ is calculated at line 12, which is the aggregate of each task's $Util_{partial\_locked}$. Finally, a boolean value suggesting if the task is allocatable.

ALGORITHM 5: Regional First Fit Decreasing (RFFD)

```
Input: $M$: set of tasks, $RM$: set of regions, $Assoc$: number of locked ways, $M_{conf}$: conflict
       matrix
Output: $IsTaskAllocatable$: boolean value
1  $RM$.sort(decreasing $R_f$);
2  while $RM$ is not empty do
3      $Success$ := false ;
4      $i$ := $RM$.front;
5      foreach $Assoc$ $a$ do
6          if $IsAllocatable(i, a, M_{conf})$ then
7              allocate region $i$ to $a$th way;
8              $Success$ := true ;
9              break ;
           end
       end
10     if $Success$ = false then
11         $unlocked\_regs := unlocked\_regs \cup i$;
       end
   end
12 $Util_{core} := \sum_{i=0}^{task\ in\ M} Util^i_{partial\_locked}$ ;
13 return $Util_{core} < 1$ ;
```

(2) Chaitin's Coloring algorithm for regions (CC): Chaitin's algorithm is used in its original form for region allocation using a conflict graph but for regions instead of task as depicted in Figure 4.
Algorithm 6 depicts the region-based task coloring mechanism, which finds non-conflicting regions that can be grouped together for a given number of colors/associativity levels. The number of colors is equal to the number of locked cache ways that can be filled within a given core. Lines 4-12 fill up two data-structures, $colorStack$ and $unlocked - regs$. Every iteration of this loop finds a region that can be placed on either of these stacks. Line 5 searches through the list of unallocated regions and finds the region with the lowest degree. A region with minimum degree is pushed onto $colorStack$ if and only if its degree is less than $Assoc$. Otherwise, the algorithm finds a task using a heuristic that focuses on minimizing a given metric. For example, in Algorithm 6 the metric $\frac{R_f}{degree}$ is minimized at line 10. 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 due to unlocking ($Util_{partial\_locked}$). Thereafter, this task is added to $unlocked\_regs$. While removing the tasks from M, we decrease the conflict $degree$ of its neighbors. A task is deemed to be allocated if the utilization bound of 1 is not exceeded. Otherwise, task allocation fails on that core. Similar to RFFD, allocation of regions is not permanent. Regions are reallocated whenever another task is being considered for allocation on the core.

As for the task selection, we use the following two heuristics:

(1) Monotonic Order (Mono): This uses task selection procedure similar to RFFD, where tasks are selected in the decreasing order of $WCET_{locked}$.

ALGORITHM 6: Region Coloring Algorithm

```
Input: $M$, $RM$, $Assoc$, $M_{conf}$ as before
Output: $IsTaskAllocatable$
1  $colorStack$ := empty;
2  $unlocked\_regs$ := empty;
3  $colorList$ := empty;
4  while $RM$ is not empty do
5      $t$ := lowest degree region by linear search of $M$ ;
6      if $t.degree < Assoc$ then
7          push $t$ onto $colorStack$ ;
8          remove $t$ from $RM$ and $M_{conf}$ ;
       end
9      else
10         $t$ := task with minimum ($\frac{R_f}{degree}$) ;
11         push $t$ onto $unlocked\_regs$ ;
12         remove $t$ from $RM$ and $M_{conf}$ ;
       end
   end
13 while $colorStack$ is not empty do
14     $t$ := Pop $colorStack$ ;
15     repopulate $M_{conf}$ ;
16     $curColor$:=0;
17     for $curColor = 0 \rightarrow Assoc - 1$ do
18         if None of the neighbors has this color then
19             $t$.color := $curColor$ ;
20             $colorList$[curColor] := $t$ ;
21             break ;
           end
       end
   end
22 $Util_{core}$ := $\sum_{i=0}^{task\ in\ M} Util^{i}_{partial\_locked}$ ;
23 return $Util_{core} < 1$ ;
```

(2) Dynamic Order (Dyn): Monotonic ordering using $Util_{locked}$ and $Util_{unlocked}$ is oblivious of the concrete number of conflicts. Here, we present a scheme that orders tasks according to their $Util_{probabilistic}$ and dynamically adjusts to changes in the regional conflict graph caused by allocations of tasks to cores. In order to obtain $Util_{probabilistic}$, we first generate a directed graph of conflicting regions. A conflicting region X is considered to be replaced by another region Y if Y's R_f is greater than that of X. The edge in the conflict graph originates in Y and points to X. This adds an incoming edge to region X and an outgoing edge from region Y. If a majority of conflicting regions replaces a region then it is considered to be unlocked for the purpose of calculating $Util_{probabilistic}$ at the task selection stage. $Util_{probabilistic}$ is obtained by the following equation:
$Util_{locked}$ + $(\sum_{i=0}^{unlocked\ regions} R_{f_i}) \times latency_{lower\ cache}$.
The task with the highest $Util_{probabilistic}$ is picked for allocation. Inspired from Chaitin's coloring algorithm, every task allocation can be used to dynamically change the state of the graph. We observe that with our greedy allocation process, when a task allocation fails, the remaining tasks are allocated to the newly created core more often than to previously existing ones. Hence, the directed graph should contain the conflicts pertaining to the unallocated regions and the regions allocated to the newly created core, only. Algorithm 7 shows the dynamic process.

The $Util_{probabilistic}$ of tasks is initialized based upon the initial graph containing all the regions. As stated earlier, the task selector picks the task t that has the largest $Util_{probabilistic}$ through a linear search over the list of tasks. Our algorithm is then called to allocate t, which changes the state of the graph as explained below.
Lines 1 through 5 try to allocate the task t to any one of the $cores$. If $TaskAllocation(t)$ succeeds, then the task is allocated to core i. Additionally, it adds t to $UnremovedTasksFromGraphs$, which is a list of allocated tasks that have not been removed from the directed graph M_{conf} and then it returns. In case $TaskAllocation(t)$ fails to allocate the task to any of the cores, lines 6 and 7 add a new core to $cores$ and allocate task t to the new core, respectively. At this point $UnremovedTasksFromGraph$ contains all the allocated tasks since the last addition of a new core. The loop starting at line 8 iterates over each of those tasks using an iterator rt. The nested loop beginning at line 9, populates rr with every iteration from the list of regions owned by rt. In line 10, one region is selected at a time, denoted as cr, which conflicts with rr. For clarification, rr and cr are regions removed from and resident within the directed graph, respectively. Lines 12 through 13 remove these edges by decrementing $NConfs$ values for the corresponding regions. If the edge is an incoming edge for cr, then line 15 decrements the incoming edge count. The ratio $\frac{numberofIncomingedges}{numberofedges}$ gives an approximate indication of whether a locked region will remain locked or not when allocated. Lines 11 and 16 compute this ratio for cr before and after removal of an edge as $priorProbability$ and $currentProbability$, respectively. If this ratio is > 0.5, then for the purpose of calculating $Util_{probabilistic}$ it is assumed to be unlocked. Thus, if this ratio changes from ≤ 0.5 to > 0.5, then the increase in utilization due to unlocking is added to $Util_{probabilistic}$ to the task that owns cr. Conversely, if the ratio changes from > 0.5 to ≤ 0.5, then $Util_{probabilistic}$ is being reduced. Lines 17 through 20 perform these changes accordingly. Thereafter, control is returned for a subsequent task to be selected based upon these changes in $Util_{probabilistic}$.

4.1. Algorithmic Complexity for Region Locking

Task-to-core allocation can be reduced to the bin packing problem, which is NP-hard [Garey and Johnson 1979], i.e., known optimal solutions are of exponential complexity. Hence, we employ heuristic approaches. We next assess the complexity of these approaches to determine their their scalable in terms of number of tasks and cores for Mono+RFFD and Dyn+CC. Let the number of tasks be X, number of regions be R, the number of cores be Y, and t be the region to be allocated next.

Mono+RFFD: This algorithm is a combination of FFD and RFFD. Here, task selection is monotonically ordered, which requires sorting with a complexity of $O(XlogX)$. The nested loops shown in Algorithm 1 (lines 2 through 10) impose a complexity of $O(XY)$. The innermost loop performs task allocation. Thus, the overall complexity is $O(XY \times O(RFFD))$. RFFD's complexity comes from the nested loops shown in Algorithm 5 from line 2 to 9. The two loops constitute a complexity of O(R) as associativity is constant. However, IsAllocatable resolves conflicts by traversing regions already allocated for an overall complexity of $O(R^2)$ for RFFD. Thus, Mono+RFFD's complexity is $O(XYR^2)$.

Dyn+CC: In order to determine the complexity of this algorithm, we refer to Algorithm 7. The outermost loop at line 2 iterates over all tasks introducing a complexity of $O(X)$. The linear search at line 3 has an overall complexity of $O(X)$ for a combined complexity of $O(X^2)$ for lines 2+3. The nested loop at line 4 iterates over each core with Chaitin's coloring-based region allocation. To derive CC's complexity, we refer to Algorithm 3. It has two loops. The first loop between lines 4 and 12 iterates over all regions.

ALGORITHM 7: Task Selection: Probabilistic Utilization

```
Input: $M_{conf}$ : Directed Conflict Graph of Regions, $InEdges$:Incoming Edges per Region,
       $NConfs$: Number of Conflicts per Region, $cores$: List of Cores,
       $UnremovedTaskFromGraphs$: Unremoved Tasks from $M_{conf}$
1  foreach $cores\ i$ do
2      if $TaskAllocation(t)isSuccessful$ then
3          allocate $t \rightarrow i$;
4          $UnremovedTaskFromGraph$ := $UnremovedTaskFromGraph \cup t$;
5          return;
       end
   end
6  $cores$:=$cores \cup new$;
7  $t \rightarrow new$;
8  foreach $UnremovedTaskFromGraph\ rt$ do
9      foreach $rt.lockedRegions\ rr$ do
10         foreach $M_{conf}\ cr$ do
11             $priorProbability$ := $\frac{InEdges[cr]}{NConfs[cr]}$;
12             $NConfs[cr]$:=$NConfs[cr]$ - 1;
13             $NConfs[rr]$:=$NConfs[rr]$ - 1;
14             if $M_{conf}[ct][rt]$==-1 then
15                 $InEdges[cr]$ := $InEdges[cr]$ - 1;
               end
16             $currentProbability$ := $\frac{InEdges[cr]}{NConfs[cr]}$;
17             if $priorProbability \leq 0.5 \wedge currentProbability > 0.5$ then
18                 $cr.task.U_{prob} = cr.task.U_{prob} + cr.U_{unlocked} - cr.U_{locked}$;
               end
19             if $priorProbability > 0.5 \wedge currentProbability \leq 0.5$ then
20                 $cr.task.U_{prob} = cr.task.U_{prob} - cr.U_{unlocked} + cr.U_{locked}$;
               end
           end
       end
   end
21 $UnremovedTaskFromGraph$:= $UnremovedTaskFromGraph \cup t$;
22 return;
```

The nested computations of the linear search at line 5, the reduction in number of conflicts for tasks conflicting with t at line 8 and 12, and the linear search at line 10 are bounded by the number of tasks, i.e., they have a complexity of $O(R)$ for a combined complexity of $O(R^2)$ for the first loop. The second loop between lines 13 and 21 iterates over all regions while popping them off the stack. The nested loop within iterates over the set of colors, which is bounded by a constant. The nested conditional at line 18 iterates over the set of neighboring nodes in the repopulated graph whose cardinality is bounded by the number of regions, R. This implies a complexity of $O(R^2)$ for the second loop, which dominates the complexity of the first loop. This is the overall complexity of Algorithm3. When combined with the complexity introduced by aforementioned loops in Algorithm 7, the overall complexity is $O(XYR^2)$. Next, we determine the complexity of the removal of regions from the conflict graph in Algorithm 7. Line 11 iterates over a set of tasks in $UremTasksFromGraphs$. This loop gets executed X number of times because each task is removed from the graph only once (mutually exclusive in one loop or the other). Whenever a task gets allocated in the loop at line 4, loop at line 11 is not executed. Thus, the combined complexity of iterations at line 2 and 11 is $O(X)$. When combined with the complexity introduced by the nested loops at line 12 and 13, the overall complexity of region removal becomes $O(XR^2)$. This is lower than $O(XYR^2)$ from above. Hence, the overall complexity of Dyn+CC is $O(XYR^2)$. With the same complexity as that of Mono+RFFD, Dyn+CC is a viable solution for task partitioning on scalable multicore architectures.

4.2. Discussion

Solutions for Scenario A can be used 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 may more attractive to developers. This allows them to select lockable lines with rough estimates of the access patterns.

Solutions for Scenario B allow us to tackle a more generic case where a task may lock a number of regions that may require portions of multiple cache ways to accommodate the locked address space for a task. These algorithms also utilize memory access frequencies, provided by static analysis tools, such as [Ramaprasad and Mueller 2011], to select regions for locking. This allows us to partition tasks while resolving conflicts among regions within a core. These solutions are also applicable to Scenario A and we present our experimental observation in the following sections.

Notice that partial locking in Scenario B does not require the WCET to be recalculated when regions are unlocked one by one as long as a processor architecture is anomaly free [Engblom 2003]. In that case, the WCET paths do not change between adjacent regions when one region is unlocked. The new WCET bound is simply calculated as

$min(WCET_{locked} + N_{refs} \times latency_{lower\ cache}, WCET_{unlocked})$

where the latencies for unlocking the region are reflected, but the unlocked WCET further serves as an upper bound if unlocking (of potentially multiple regions) results in large aggregates of additional latencies. Conversely, an architecture with timing anomalies requires either (a) recalculation of the WCET after each region is unlocked as paths before/after this regions may change or (b) our heuristic is applied based on R_f and only the final set of unlocked regions is subjected to another WCET analysis to accurately update utilizations and this avoid over-allocation. While (a) is costly in terms of WCET recalculations, (b) may result in slight degradations of the heuristics, but the result is still validated and $WCET_{unlocked}$ with all regions unlocked always provides a safe bound (as given above), i.e., in the theoretical case that a partially unlocked scenarios exceeds $WCET_{unlocked}$ due to anomalies, one can simply unlock all regions to obtain $WCET_{unlocked}$ in the worst case.

5. TASK SET GENERATION

Due to the unavailability of a full-blown real-time application for massive multicore architectures, we decided to utilize synthetic task sets in our experiments. This allows us to vary task set parameters like utilization of tasks, size of the locked regions, number of tasks, and number of regions per task, which in turn tests corner cases of our algorithms. For scenario B, as it allows larger numbers of locked regions, we also focus upon generation of denser conflict graphs. We assume that static analysis tools, such as [Ramaprasad and Mueller 2011], deliver the $WCET_{locked}$, $WCET_{unlocked}$ and R_f, which is orthogonal and beyond the scope of this work.

Here, we explain the procedure of task-set generation for Scenario B, as it is less restrictive in its constraints.

(1) We use Task Graph For Free (TGFF) [Dick et al. 1998] to generate a conflict graph with a given number of regions (200) and with a randomly chosen number of con-

flicts per region, which is a randomly generated proportion of the total number of regions. (For high conflicts, the upper bound is 0.9 and lower bound varies from 0.1-0.5; for low conflicts, the lower bound is 0.1 and the upper bound varies from 0.2-0.5).

(2) We randomly generate a number of accesses for each region within a given range of accesses (50-200).

(3) We generate tasks and randomly pick a number of regions (2-8) for each until all the regions have been allocated to a task. The last task generated may have a lower number of regions than the lower bound of two regions. The lockable associativity of the L1 cache is assumed to be four.

(4) The total number of references to locked regions 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 randomly choose 6-9 instructions per load. This lets us infer the number of instruction fetches that incur L1 cache hits (see Section 2). These assumptions allow us to derive a $WCET_{locked}$ for a task. The processor cores are assumed to feature an in-order 3-stage pipeline, with each instruction taking one cycle to execute, except branch instructions which incur a penalty of three cycles. The L1, L2 and Memory access latencies have been assumed to be 1, 10 and 100 cycles, respectively.

(5) 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. 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.

(6) Next, we assign periods to each task i to group them into different utilization categories: medium-high utilization $(0.55 > u_{locked i} > 0.30)$, and medium-low utilization $(0.30 > u_{locked i} > 0.1))$.

(7) We assume that the tasks do not have any inter-task dependencies.

(8) We assume the utilization of tasks to be equal to their density. In other words, a task's deadline is equal to its period.

The generation of benchmarks for Scenario A differs from Scenario B in the following aspects:

(1) First, a given number of tasks are generated. Then, a number of randomly generated locked cache regions is assigned to each of them instead of generating a conflict graph. These locked cache regions are generated such that there is no intra-task conflict. In order to generate memory regions, we assume the cache architecture shown in Table II, which is loosely resembling parameters similar to the Tilera architecture [Tilera 2009]. The table also displays the characteristics of the tasks and locked regions generated. The lockable associativity of cache in Scenario A is lower than that of scenario B.

(2) In Scenario A, we unlock a task; in Scenario B, we unlock a region. In order to observe the performance of task sets that have only high utilization, we partition the utilization range into: high $0.55 < U_{locked} < 0.40$, medium $0.4 < U_{locked} < 0.25$, low $0.1 < U_{locked} < 0.25$.

Table II. System Parameters

Parameter	Value
Cache Line Size	32B
L1 Cache Size/Associativity	8KB/2-way
Lockable associativity	1/2
locked regions per task	1 - 4
Sets locked by a task	8-114 out of 128
Size of locked regions	8-57 sets
Max. size of task sets	42
total tasks generated	126
Min. locked regions by a task	1

6. EVALUATION

This section firstly presents the improvement of cache-aware schemes over cache-unaware schemes for task sets from Scenario A. It also compares the performance of GFFD and CoFFD for task sets of Scenario A. This is followed by evaluation with different combinations of Task Selection and Task Allocation (Mono+RFFD, Mono+CC, Dyn+RFFD, Dyn+CC) on task sets of Scenario B. We then compare the performance of the best solutions from both the scenarios when applied upon task sets of Scenario A. This assesses the applicability of these algorithms.

6.1. Scenario A

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 III. Allocated Cores for Cache-aware & Cache-unaware Schemes, Unlocking Allowed when Adding Cores

Number	High Util.		Med. Util.		Low Util.	
of Tasks	Unaware	Aware	Unaware	Aware	Unaware	Aware
4	4	**3**	4	**2**	3	**2**
8	8	**5**	8	**4**	5	**3**
12	12	**8**	12	**5**	8	**4**
16	16	**10**	16	**8**	12	**6**
20	20	**13**	20	**11**	16	**8**
24	24	**15**	23	**15**	19	**10**
28	28	**19**	27	**19**	21	**11**
32	32	**20**	31	**21**	22	**12**
36	36	**21**	35	**22**	23	**15**
42	42	**25**	41	**24**	24	**17**

Cache-unaware vs. Cache-aware: First, we compare the cache-unaware schemes (FFD, NFFD) against cache-aware ones (GFFD, CoFFD). Table III shows the best allocations produced by schemes within the two categories, i.e., NFFD (cache-unaware) and CoFFD (cache-aware). Under this policy, tasks are unlocked when the number of cores has to be increased to fit all tasks, and then relocked according to the respective algorithm. On average, the number of cores used by cache-aware schemes is 40% less than that of contemporary allocation schemes applicable for distributed core mechanisms. We also observe that the contemporary FFD fails to allocate high utilization task sets. It performs worse than NFFD for low utilization task sets as shown earlier in Table 1.

Allocations while retaining locked state: Table IV depicts the results of our algorithms when tasks are allocated in locked state, i.e., with an execution time of $WCET_{locked}$. Under this policy, tasks remain locked when the number of cores has to be

Table IV. Allocated Cores for CoFFD & GFFD: All Tasks Locked, no Unlocking when Adding Cores

Number	**High Util.**		**Med. Util.**		**Low Util.**	
of Tasks	**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**

increased to fit the remaining tasks. The first column shows the number of tasks in the task set. The second and third columns show the number of cores allocated by GFFD and CoFFD, respectively, when a task set is composed of high utilization tasks only. The fourth and fifth 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 avoids conflicts strategically due to its coloring scheme while the greedy scheme results in a less conflict-conscious allocation.

Table V. 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 V 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 utilization can be allocated 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 also in improving system utilization by over 18% for task sets with large numbers of tasks. This is because GFFD inflates the execution budget of tasks 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 combined heuristics for selecting spilled tasks. 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 4). 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 V 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.

6.2. Scenario B

Next we present results of our algorithms that statically partition tasks for task sets of Scenario B. For scenario B, we generated a large set of task sets that can be varied with regard to utilization (medium-high, medium-low and mixed) and density of conflicts (high and low). A total of 1200 experiments were conducted. For each conflict ratio and utilization range, 10 task sets were created with different randomization seed values.

Table VI. Mono+RFFD vs. Mono+CC for High Conflict

Conflict Ratio Range	Performance Category	high-medium Util. Range		low-medium Util. Range		mixed Util. Range	
		Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC
0.1-0.9	Best	24	**22**	14	**12**	19	**17**
	Average	22.4	**21.3**	12.9	**12**	18.4	**17.1**
	Worst	**22**	**22**	**12**	**12**	**18**	**18**
0.2-0.9	Best	25	**22**	14	**12**	17	**15**
	Average	22.6	**21.2**	13	**12.1**	18.1	**17**
	Worst	**21**	**21**	**13**	**13**	19	**18**
0.3-0.9	Best	27	**23**	14	**12**	18	**15**
	Average	23.7	**22**	13.6	**12.5**	19	**17.6**
	Worst	21	**20**	13	**12**	**19**	**19**
0.4-0.9	Best	23	**21**	15	**13**	20	**17**
	Average	24.1	**22.8**	13.5	**12.6**	19	**17.5**
	Worst	**21**	**21**	**12**	**12**	**17**	**17**
0.5-0.9	Best	27	**23**	15	**13**	20	**17**
	Average	25	**23.1**	14	**13.2**	19.3	**18.1**
	Worst	**23**	**23**	**13**	**13**	**19**	**19**

Table VII. Mono+RFFD vs. Mono+CC for low Conflict

Conflict Ratio Range	Performance Category	high-medium Util. Range		low-medium Util. Range		mixed Util. Range	
		Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC
0.1-0.2	Best	21	**20**	10	**9**	15	**14**
	Average	20.2	**20.1**	9.8	**9.7**	14.1	**13.9**
	Worst	20.2	20.1	9.8	9.7	**13**	**13**
0.1-0.3	Best	**20**	21	12	**11**	16	**15**
	Average	**20.3**	20.4	10.8	**10.6**	15.3	**15.1**
	Worst	20.3	20.4	**10**	11	**16**	17
0.1-0.4	Best	22	**21**	12	**10**	15	**14**
	Average	**20.1**	**20.1**	11.2	**10.3**	15.2	**14.9**
	Worst	**18**	19	**11**	**11**	**14**	15

Mono+RFFD vs. Mono+CC: Table VI shows the impact of CC over RFFD on high conflict task sets. The first column shows the conflict ratio range of the high conflict benchmarks. Due to the large set of results, we only present the best and worst cases for CC over RFFD along with the average number of cores used per benchmark. The first column indicates conflict ratio ranges. The second column distinguishes best-, average and worst-case allocation results. The third, fourth and fifth columns show the results associated with benchmarks of high-medium, low-medium and mixed utilization ranges. Each of these columns have two sub-columns that depict the results for Mono+RFFD and Mono+CC for each utilization range and case. The results show that Mono+CC allocates the task sets to fewer or a matching number of cores compared to Mono+RFFD for all the high-conflict benchmarks. This is primarily because the conflicts are high enough such that resolving them locally at the level of a core proves to be useful. Table VII compares the performance of CC over RFFD on low conflict task sets. The layout of the table is the same as that of Table VI. We observe here that the worst cases force Mono+CC to map the tasks onto more cores than required by Mono+RFFD. This highlights the limits of a locally efficient coloring mechanism as it is inferior at a global scale. This is because even though the coloring scheme does a better job at packing a given set of tasks within a core, sometimes failure to allocate some tasks within a core could pave the path for a better fit of subsequent tasks. This is the basis for our Dyn algorithm that is sensitive to conflicts and provides better global ordering during task selection.

Table VIII. Average allocations performed by Scenario B algorithms

Conflict Ratio Range	high-medium Util				low-medium Util				mixed Util			
	Mono+ RFFD	Mono+ CC	Dyn+ RFFD	Dyn+ CC	Mono+ RFFD	Mono+ CC	Dyn+ RFFD	Dyn+ CC	Mono+ RFFD	Mono+ CC	Dyn+ RFFD	Dyn+ CC
0.1-0.9	22.4	21.3	21.1	**20.3**	12.9	12	12.4	**11.6**	18.4	17.1	16.6	**16.1**
0.2-0.9	22.6	21.2	21	**20.4**	13	12.1	12.3	**11.7**	18.1	17	16.4	**15.9**
0.3-0.9	23.7	22	22.2	**21**	13.6	12.5	12.9	**12.2**	19	17.6	17.5	**16.7**
0.4-0.9	24.1	22.8	22.5	**21.3**	13.5	12.6	12.8	**12.3**	19	17.5	17.3	**16.8**
0.5-0.9	25	23.1	24.1	**22.1**	14	13.2	13.2	**12.4**	19.3	18.1	17.9	**17.1**
0.1-0.2	20.2	20.1	19.8	**19.7**	9.8	9.7	9.7	**9.4**	14.1	13.9	13.4	**13.4**
0.1-0.3	20.3	20.4	20.1	**20**	10.8	10.6	10.5	**10.3**	15.3	15.1	14.3	**14.1**
0.1-0.4	20.1	20.1	19.9	**19.7**	11.2	10.3	10.6	**10.2**	15.2	14.9	14.4	**14.2**

Table IX. Best and Average improvement by Dyn+CC over Mono+RFFD

Conflict Ratio Type	Improvement Case	high-medium Util	low-medium Util	mixed util
High	Best	22% 21 vs. 27	17% 10 vs. 12	19% 17 vs. 21
	Average	10.49%	10.03%	11.81%
Low	Best	5.26% 18 vs. 19	17% 10 vs. 12	13% 13 vs. 15
	Average	1.78%	5.72%	6.21%

Overall, the best case is when the Mono+CC outperforms Mono+RFFD by the greatest margin for a single benchmark. The worst case is when Mono+CC is either outperformed by Mono+RFFD or the margin by which the Mono+CC is better than

Mono+RFFD is the least for a single benchmark. On average, the number of cores allocated for a set of benchmarks is classified under a conflict ratio or utilization range.

Scenario B algorithms performance: Table VIII shows the average performance of the various algorithmic combinations of the task selection and task allocation algorithms for all conflict ratio ranges (low and high) on all utilization ranges (high-medium, low-medium and mixed). The first column shows the conflict ratios. The second, third and fourth columns show the results on benchmarks with high-medium, low-medium and mixed utilization ranges, respectively. Each of those columns have 4 sub-columns with task selection/allocation combinations (in the following order: Mono+RFFD, Mono+CC, Dyn+RFFD, Dyn+CC). Due to the large number of results, we again resort to presenting the average number of cores allocated per benchmark for given conflict and utilization ranges. The highlighted numbers are again the best allocations. The highlighted results clearly show that Dyn+CC produces the best allocations compared to any other combination. Also, Dyn+RFFD consistently outperforms Mono+RFFD. Dyn+RFFD performs better than Mono+CC for the cases where the latter performed worse than Mono+RFFD. This shows the effectiveness of the Dynamic task selection mechanism. Note that Dyn+RFFD does not always outperform Mono+CC, even though Dyn+CC is the best performing algorithm overall.

One should note that Dyn+RFFD does not always outperform Mono+CC, even though Dyn+CC is the best performing algorithm overall. Table IX shows the improvement achieved by Dyn+CC over the base case of Mono+RFFD. The first column shows the two conflict ratio range types. The second column identifies the best and average case improvements in each of the cases. The third, fourth and fifth columns depict the improvement in terms of percentages, while the best case also shows the allocated core numbers (Dyn+CC vs. Mono+RFFD). Dyn+CC shows distinct high improvement percentage benefits for high conflict task sets (up to 6 cores). Among all the utilization ranges, the mixed utilization range produces the highest average improvement. This shows that the Dyn+CC algorithm not only works well with corner cases but it also performs best with systems that have a variety of utilizations and conflict densities.

Table X. CoFFD vs. Dyn+CC: Task vs. Region Unlocking

Number	**High Util.**		**Med. Util.**		**Low Util.**	
of Tasks	**CoFFD**	**Dyn+CC**	**CoFFD**	**Dyn+CC**	**CoFFD**	**Dyn+CC**
16	**10**	**10**	**8**	**8**	**6**	**6**
20	**13**	14	11	**10**	**8**	**8**
24	**15**	17	15	**12**	**10**	**10**
28	**19**	20	19	**14**	**11**	**11**
32	**20**	22	21	**15**	**12**	13
36	**21**	25	22	**18**	**15**	**15**
42	**25**	27	24	**20**	**17**	**17**

CoFFD vs. Dyn+CC: Dyn+CC has been the most useful combinations for task sets of Scenario B. Thus, it becomes imperative to gauge its effectiveness relative to task sets of Scenario A and compare its performance against CoFFD. Table X depicts such results with higher conflict density task sets. CoFFD performs better that Dyn+CC for high utilization tasks. That is because unlocking a region or a task does not allow multiple tasks to be scheduled together. Thus, the global ordering achieved by CoFFD is better. However, CoFFD is unable to unlock tasks of medium utilization either where the unlocking at regional granularity is feasible. Thus, Dyn+CC performs better. With low utilization task sets, CoFFD is able to unlock tasks and still allocate other tasks along with it. This makes both CoFFD and Dyn+CC perform well. Nonetheless, CoFFD benefits from a superior global ordering, which allows it to find a better allocation with our 32-tasks set as shown in the table.

7. RELATED WORK

In the 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 [Puaut and Decotigny 2002; Puaut 2006]. Several methods have been developed to lock program data that is hard to analyze statically [Vera et al. 2003]. Further techniques have been developed for cache locking that provide performance comparable to that obtained with scratchpad allocation [Puaut and Pais 2007]. Recently, cache locking has also been proposed for multi-core systems that use shared L2 caches [Suhendra and Mitra 2008]. Liu et al. propose cache locking for private L1 caches while using cache partitioning for L2 caches [Liu et al. 2010]. Their focus has been upon reducing the task utilization while partitioning a task set on all the processor cores in the system. This is applicable to unscalable shared cache architectures. In contrast, our work focuses upon optimizing allocation of computational resources. These related efforts show that cache locking is a viable solution in future real-time system designs for multicores. Guan et al. propose L2 cache partitioning using cache coloring for soft real-time systems [Guan et al. 2009]. Paolieri et al. have proposed hardware cache partitioning mechanisms for multicore real-time systems with shared L2 caches [Paolieri et al. 2011]. However, the focus of both these cache partitioning schemes is to pack as many real-time tasks as possible on an unscalable multicore architecture. Their handling of a set of WCET bounds for different partition sizes of a matrix of WCETs is search based. A breadth first search is followed by sorting results according to variance across partition sizes, which are then first-fit allocated in a greedy manner of a most sensitive task with non-sensitive others. Their concept of sensitivity is agnostic of the causes of high WCET variance (close to GFFD) while our coloring approach (CoFFD) explicitly detects the causes of conflicts and prioritizes allocations accordingly.

Choffnes *et al.* have proposed migration policies for multicore fair-share scheduling [Choffnes et al. 2008]. 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 [Anderson et al. 2006; Calandrino and Anderson 2008]. 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 [Li et al. 2007; Li et al. 2008]. 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.* [Eisley et al. 2008] develop a cache capacity increasing scheme for multicores that scavenges unused neighboring cache lines.

Paolieri *et al.* [Paolieri et al. 2009] 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.* [Ouyang and Xie 2010] have proposed extending Quality of Service support to mesh-based interconnects but their study is limited to the on-chip network traffic.

8. CONCLUSIONS

The use of multicore 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. This work studies allocation of real-time tasks with locked caches on distributed cache systems. Contemporary static scheduling schemes may not use locked caches. However, this renders certain high utilization tasks unschedulable as their unlocked WCET is prohibitively high. A simplistic solution would be to allowing locking of such tasks and placing locked tasks onto different cores. We call this Naive locked FFD (NFFD) as it locks certain tasks with high utilization and is cache-unaware.

This paper proposes two cache-aware algorithms for Scenario A type task sets. These algorithms allocate tasks in a multicore environment where tasks are allowed to lock cache lines in a specified subset of cache ways in each core's private L1 cache. The first algorithm, GFFD, is an enhanced version of the First Fit Decreasing (FFD) algorithm. The second, CoFFD, is based on a graph coloring method. CoFFD reduces the number of core requirements from 25% to 60% compared to NFFD with an average reduction of 40%. CoFFD consistently performs better than GFFD as it lowers both the number of cores and system utilization.

This paper further presents algorithms for task sets of Scenario B. These task sets may have intra-task cache conflicts. Conflicting locked regions are allocated to different cache ways. These algorithms resolve conflicts at region level by locking and unlocking regions instead of locking or unlocking entire tasks, as was the case in Scenario A. Here, task partitioning is split into task selection and task allocation phases. We use two task allocation mechanisms, namely (a) Regional FFD and (b) Chaitin's coloring. We propose two task selection algorithms, namely (a) Monotone and (b) Dynamic. The combination of Mono+CC outperforms Mono+RFFD for highly conflicted task sets. This shows the effectiveness of using the coloring mechanism at individual cores. In contrast, Mono+CC does not consistently perform the best for low contention task sets.

This necessitates a global ordering scheme like Dyn, which complements core level coloring. Our results show that Dyn+CC consistently performs better than Mono+RFFD. For high contention task sets, Dyn+CC achieves up to a 22% reduction in the number of cores allocated, i.e., it allocates 21 cores as opposed to 27 for Mono+RFFD. Even for low contention task sets, Dyn+CC is able to achieve a reduction of up to 17%. Since Dyn+CC deals with a more generic problem set, it is also applicable to task sets of Scenario A. While comparing CoFFD against Dyn+CC, we observe that CoFFD performs better than Dyn+CC for high utilization tasks. That is because unlocking a region or a task does not allow multiple tasks to be scheduled together. Thus, the global ordering achieved by CoFFD is superior. However, CoFFD is unable to unlock medium utilization tasks. In contrast, unlocking at regional granularity is feasible and thus Dyn+CC performs better.

These observations suggest that CoFFD and Dyn+CC perform far better than contemporary FFD-based task partitioning on distributed core CMPs. Overall, this work is unique in considering the challenges of future multicore architectures for real-time systems. It provides key insights into task partitioning with locked caches for architectures with private caches.

REFERENCES

ADAPTEVA. Parallella computer specifications.

AKESSON, B., GOOSSENS, K., AND RINGHOFER, M. 2007. Predator: A predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '07. 251–256.

ANDERSON, J., CALANDRINO, J., AND DEVI, U. 2006. Real-time scheduling on multicore platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 179–190.

ARM. Arm11 mpcore processor.

BURCHARD, A., LIEBEHERR, J., OH, Y., AND SON, S. 1995. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. on Computers 44,* 12, 1429–1442.

BUSQUETS-MATRAIX, J. V. 1996. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*.

BUSQUETS-MATRAIX, J. V. 1997. Hybrid instruction cache partitioning for preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*. 56–63.

CALANDRINO, J. AND ANDERSON, J. 2008. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Euromicro Conference on Real-Time Systems*. 209–308.

CHAITIN, G. J. 1982. Register allocation & spilling via graph coloring. *SIGPLAN Not. 17,* 6, 98–101.

CHATTOPADHYAY, S., ROYCHOUDHURY, A., AND MITRA, T. 2010. Modeling shared cache and bus in multicores for timing analysis. In *Proceedings of the 13th International Workshop on Software &#38; Compilers for Embedded Systems*. SCOPES '10. ACM, New York, NY, USA, 6:1–6:10.

CHOFFNES, D., ASTLEY, M., AND WARD, M. J. 2008. Migration policies for multi-core fair-share scheduling. *ACM SIGOPS Operating Systems Review 42*, 92–93.

DE DINECHIN, B. D., DE MASSAS, P. G., LAGER, G., LGER, C., ORGOGOZO, B., REYBERT, J., AND STRUDEL, T. 2013. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science 18,* 0, 1654 – 1663. 2013 International Conference on Computational Science.

DICK, R. P., RHODES, D. L., AND WOLF, W. 1998. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*. CODES/CASHE '98. IEEE Computer Society, Washington, DC, USA, 97–101.

EISLEY, N., PEH, L.-S., AND SHANG, L. 2008. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. PACT '08. ACM, New York, NY, USA, 197–207.

ENGBLOM, J. 2003. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 152.

FREESCALE. P4080 multicore processor.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

GUAN, N., STIGGE, M., YI, W., AND YU, G. 2009. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*. EMSOFT '09. ACM, New York, NY, USA, 245–254.

HARDY, D., PIQUET, T., AND PUAUT, I. 2009. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. RTSS '09. IEEE Computer Society, Washington, DC, USA, 68–77.

HERTER, J., BACKES, P., HAUPENTHAL, F., AND REINEKE, J. 2011. Cama: A predictable cache-aware memory allocator. In *Euromicro Conference on Real-Time Systems*. 23–32.

HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., WILSON, H., BORKAR, N., SCHROM, G., ET AL. 2010. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 108–109.

LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. SC '07. ACM, New York, NY, USA, 53:1–53:11.

LI, T., BRETT, P., HOHLT, B., KNAUERHASE, R., MCELDERRY, S., AND HAHN, S. 2008. Operating system support for shared-isa asymmetric multi-core architectures. In *Workshop on the Interaction between Operating Systems and Computer Architecture*. 19–26.

LIEDKE, J., HÄRTIG, H., AND HOHMUTH, M. 1997. Os-controlled cache predictability for real-time systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 213–223.

LIU, T., LI, M., AND XUE, C. J. 2009. Minimizing wcet for real-time embedded systems via static instruction cache locking. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 35–44.

LIU, T., ZHAO, Y., LI, M., AND XUE, C. J. 2010. Task assignment with cache partitioning and locking for wcet minimization on mpsoc. In *International Conference on Parallel Processing*. 573–582.

MANCUSO, R., DUDKO, R., BETTI, E., CESATI, M., CACCAMO, M., AND PELLIZZONI, R. 2013. Real-time cache management framework for multi-core architectures. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 45–54.

MUELLER, F. 1995. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*. 137–145.

OUYANG, J. AND XIE, Y. 2010. Loft: A high performance network-on-chip providing quality-of-service support. *Microarchitecture, IEEE/ACM International Symposium on 0*, 409–420.

PAOLIERI, M., QUIÑONES, E., CAZORLA, F. J., BERNAT, G., AND VALERO, M. 2009. Hardware support for wcet analysis of hard real-time multicore systems. In *Proceedings of the 36th annual international symposium on Computer architecture*. ISCA '09. ACM, New York, NY, USA, 57–68.

PAOLIERI, M., QUIÑONES, E., CAZORLA, F. J., DAVIS, R. I., AND VALERO, M. 2011. Ia3: An interference aware allocation algorithm for multicore hard real-time systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 280–290.

PLAZAR, S., KLEINSORGE, J. C., MARWEDEL, P., AND FALK, H. 2012. Wcet-aware static locking of instruction caches. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 44–52.

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

PUAUT, I. AND DECOTIGNY, D. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*. RTSS '02. IEEE Computer Society, Washington, DC, USA, 114–.

PUAUT, I. AND HARDY, D. 2007. Predictable paging in real-time systems: A compiler approach. In *Euromicro Conference on Real-Time Systems*. 169–178.

PUAUT, I. AND PAIS, C. 2007. 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*. EDA Consortium, San Jose, CA, USA, 1484–1489.

RAMAPRASAD, H. AND MUELLER, F. 2011. Tightening the bounds on feasible preemptions. *ACM Trans. Embed. Comput. Syst. 10,* 2, 27:1–27:34.

SARKAR, A., MUELLER, F., AND RAMAPRASAD, H. 2012. Static task partitioning for locked caches in multicore real-time systems. In *Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 161–170.

SUHENDRA, V. AND MITRA, T. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*. ACM, New York, NY, USA, 300–303.

TILERA. 2009. Tilera processor family. http://www.tilera.com/.

VERA, X., LISPER, B., AND XUE, J. 2003. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*. RTSS '03. IEEE Computer Society, Washington, DC, USA, 154–.

VERA, X., LISPER, B., AND XUE, J. 2007. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst. 7,* 1, 4:1–4:38.

WARD, B., HERMAN, J., KENNA, C., AND ANDERSON, J. 2013. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*. 157–167.

WOLFE, A. 1993. Software-based cache partitioning for real-time applications. In *Workshop on Responsive Computer Systems*.

YUNY, H., MANCUSOZ, R., WU, Z.-P., AND PELLIZZONI, R. 2014. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*.