An Optimal Semi-Partitioned Scheduler
Assuming Arbitrary Affinity Masks

Sergey Voronov*, James H. Anderson†, Department of Computer Science, University of North Carolina at Chapel Hill, United States
Email: *rdkl@cs.unc.edu, †anderson@cs.unc.edu

Abstract—Modern operating systems allow task migrations to be restricted by specifying per-task processor affinity masks. Such a mask specifies the set of processor cores upon which a task can be scheduled. In this paper, a semi-partitioned scheduler, AM-Red (affinity mask reduction), is presented for scheduling implicit-deadline sporadic tasks with arbitrary affinity masks on an identical multiprocessor. AM-Red is obtained by applying an affinity-mask-reduction method that produces affinities in accordance with those specified, without compromising feasibility, but with only a linear number of migrating tasks. It functions by employing a tunable frame of size $|F|$. For any choice of $|F|$, AM-Red is soft-real-time optimal, with tardiness bounded by $|F|$, but the frequency of task migrations is proportional to $|F|$. If $|F|$ divides all task periods, then AM-Red is also hard-real-time-optimal (tardiness is zero). AM-Red is the first optimal scheduler proposed for arbitrary affinity masks without future knowledge of all job releases. Experiments are presented that show that AM-Red is implementable with low overhead and yields reasonable tardiness and task-migration frequency.

Index Terms—scheduling theory, multicore, processor affinity masks

I. INTRODUCTION

On multicore machines, particularly ones with relatively high core counts, it is often desirable to limit task migrations to lessen cache- and I/O-related overheads [13] and to enable load balancing [28], [32], among other reasons [24]. Processor affinity masks are an operating-system (OS) mechanism that enables allowed migrations to be flexibly determined. A given task’s affinity mask specifies which cores it is allowed to execute upon. General-purpose OSs that support affinity masks include Windows, Linux, and MacOS X. Real-time OSs (RTOSs) that support them include FreeRTOS [21], QNX [2], VxWorks [3], and many others.

Unfortunately, in the real-time systems domain, no non-clairvoyant optimal scheduler has heretofore been proposed that allows arbitrary affinity masks. Thus, while OSs provide flexible control over migrations through affinity masks in theory, such support must typically be restricted in practice. For example, under Linux’s SCHED_DEADLINE scheduler [12], [20], affinity masks are essentially ignored: any task is assumed to be executable on any core [4]. Such conservative behavior can be changed, but doing so requires disabled admission control, and no response-time guarantees can be provided.

In this paper, we show for the first time that the goals of allowing arbitrary affinity masks and scheduling real-time tasks optimally do not fundamentally conflict. We do so by presenting a new scheduler, AM-Red (affinity mask reduction), that optimally schedules implicit-deadline sporadic tasks. Before delving into notable specifics concerning AM-Red, we first review relevant prior work to provide context.

Related work. The existing literature pertaining to scheduling real-time tasks with affinity masks is not very extensive. For hard real-time (HRT) implicit-deadline sporadic task systems with arbitrary affinity masks, Baruah et al. [7] proposed an exact feasibility test and corresponding scheduler. However, their scheduler has an offline phase with high time complexity and is a fluid scheduler that gives rise to unboundedly frequent task migrations, which (seemingly) can be reduced to a practical level only through clairvoyant knowledge of all job releases. Muneeswari et al. [23] presented a scheduler supporting affinities that they claimed is applicable to real-time systems, but they provided no analysis to support this claim. Cerqueira et al. [11] presented a fixed-priority arbitrary-affinity scheduler, but it is not optimal. Gujarati et al. [15], [16] presented several schedulability tests for any job-level fixed-priority scheduler assuming arbitrary affinities, but these tests are all non-tight or non-polynomial.

Hierarchical affinity masks are often used on multicore machines to reflect multi-level cache hierarchies (L1, L2, etc.). With hierarchical affinities, if the masks of two tasks intersect, then one must be contained within the other. For hierarchical affinities, Bonifaci et al. [8] proposed an HRT scheduler (which evolved from a prior approach [11]) that ensures a certain “greedy” property that avoids wasted processing capacity. However, they provided no schedulability test.

Contributions. The main contribution of this paper is AM-Red, a new scheduling algorithm for implicit-deadline sporadic task systems with arbitrary affinity masks. AM-Red is a semi-partitioned scheduler; under such schedulers, only certain tasks are allowed to migrate and these tasks are determined in an offline allocation phase [5].

AM-Red schedules tasks by iteratively considering a schedule computed offline for a window of time called a frame. The frame size $|F|$ is a configurable parameter. For soft real-time (SRT) task systems that require bounded deadline tardiness, AM-Red is optimal and ensures a tardiness bound of $|F|$. The frame size $|F|$ also determines the frequency of task migrations, so choosing $|F|$ yields a tradeoff: larger values reduce migration costs while smaller values reduce tardiness. If $|F|$ divides all task periods, then AM-Red is also optimal for scheduling HRT task systems. For $n$ tasks executing on $m$
processors, if masks are hierarchical, then AM-Red requires $O(m + n)$ time complexity for its offline phase and $O(1)$ time per scheduling decision; these time bounds are asymptotically optimal. To the best of our knowledge, AM-Red is the first non-clairvoyant scheduler to be proposed that is HRT/SRT-optimal for implicit-deadline sporadic tasks under arbitrary affinity masks.

In addition to presenting AM-Red, we also explore a number of issues concerning affinity-mask reductions. In particular, we consider affinity graphs that aggregate the specified masks of all tasks, and present a method that can reduce the number of edges in such a graph without compromising task-system feasibility. While feasibility can be assessed using the test of Baruah et al. [7], we instead use a test proposed here that has lower time complexity. Our reduction method yields affinity masks under which at most $(m - 1)$ tasks migrate. This property is actually instrumental in enabling a semi-partitioned approach.

In order to assess the efficacy of AM-Red, we recorded scheduling and other OS overheads under it in an actual LITMUS$^RT$ implementation on a 24-core Intel platform. We found that these overheads tended to be small, on the order of just a few microseconds. We also experimentally compared AM-Red to the (non-optimal) scheduler proposed in [8] on the basis of migration frequency and tardiness. In these experiments, AM-Red demonstrated performance even better than our analysis predicts. It proved capable of ensuring tardiness of just a few tens of milliseconds with task-migration frequencies that were nearly two orders of magnitude less than those under the scheduler from [8] in some cases. Recall that, unlike AM-Red, that scheduler is limited to hierarchical masks and has no schedulability test.

**Organization.** In the rest of this paper, we provide needed background (Sec. II), present our new feasibility test (Sec. III), develop algorithm AM-Red by considering first “acyclic” affinities (Sec. IV) and then arbitrary ones (Sec. V), consider the special case of hierarchical affinities (Sec. VI), and conclude (Sec. VIII).

II. BACKGROUND

We consider the problem of scheduling $n$ implicit-deadline sporadic tasks, $τ_1, ..., τ_n$, on $m$ identical unit-speed cores, $π_1, ..., π_m$. We assume familiarity with the implicit-deadline sporadic task model, consider only task sets in accordance with this model, and assume that all time-related parameters are rational.¹ We will use the following notation: $C_i$ denotes the worst-case execution time of task $τ_i$, $T_i$ denotes its period, $D_i = T_i$ denotes its relative deadline, and $U_i = C_i / T_i \leq 1$ denotes its utilization; $J_{i,k}$ denotes the $k^{th}$ job released by $τ_i$, and $C_{i,k} \leq C_i$ denotes the execution time of $J_{i,k}$; $U = \sum U_i$ denotes the total system utilization. Job $J_{i,k}$ has an absolute deadline $d_i$ that occurs $D_i$ time units after its release, and once it has received a processor allocation equal to $C_{i,k}$, it is completed. Job $J_{i,k+1}$ cannot be scheduled until the prior job of $τ_i$, $J_{i,k}$, has completed, even if $J_{i,k}$ misses its (absolute) deadline. As is typical in scheduling-theoretic work, we assume that overheads are negligible (though we do examine measured overheads under AM-Red in Sec. VII).

If a job has a deadline at time $t_d$ and completes at time $t_c$, then its tardiness is defined as $\max(0, t_c - t_d)$. The tardiness of task $τ_i$ is the supremum of the tardiness of any of its jobs. If this value is finite, then we say that $τ_i$ has bounded tardiness.

A task set $τ$ is HRT-schedulable (resp., SRT-schedulable) under scheduling algorithm $S$ if each task in $τ$ has zero (resp., bounded) tardiness in any schedule for $τ$ generated by $S$. A task set $τ$ is HRT-feasible (resp., SRT-feasible) if, for any job-release sequence (as allowed by the sporadic task model), a schedule exists in which each task has zero (resp., bounded) tardiness. Scheduling algorithm $S$ is HRT-optimal (resp., SRT-optimal) if every HRT-feasible (resp., SRT-feasible) task set $τ$ is HRT-schedulable (resp., SRT-schedulable) under $S$. Although HRT- and SRT-feasibility are fundamentally different concepts in some contexts, we show later that in the context of this paper, they are actually equivalent.

**Affinity masks.** In practice, affinity masks are usually specified using bit-vectors, but we opt for a more abstract specification. In particular, we define the affinity mask $α_i$ of task $τ_i$ to be the set of cores upon which $τ_i$ is allowed to execute. We define the aggregate affinity mask of a subset of tasks $τ' \subseteq τ$ as $α_{τ'} = \bigcup_{τ_i \in τ'} α_i$. We call the aggregate affinity mask $α_τ$ of the set of all tasks $τ$ the system affinity mask.

For a given task set $τ$, we define a bipartite undirected graph called an affinity graph, denoted $AG(τ)$, as follows: $AG(τ)$ has $n$ vertices $τ_1, ..., τ_n$ (representing tasks), and $m$ vertices $π_1, ..., π_m$ (representing cores), and contains edge $(τ_i, π_j)$ if and only if $π_j \in α_i$ (i.e., task $τ_i$ can execute on core $π_j$).

**Example 1.** Consider a task set $τ$ with four tasks, $τ_1, ..., τ_4$, to be scheduled on three cores, $π_1, π_2, π_3$, with affinity masks $α_1 = \{π_1\}$, $α_2 = \{π_1\}$, $α_3 = \{π_1, π_2\}$, and $α_4 = \{π_2, π_3\}$. $AG(τ)$ is shown in Fig. [1a]. The aggregate affinity mask for $\{τ_1, τ_4\}$ is $\{π_1, π_2, π_3\}$, while for $\{τ_2, τ_3\}$ it is $\{π_1, π_2\}$.

**Important affinity-mask categories.** For a given task set $τ$, we call $α_τ$ acyclic if and only if $AG(τ)$ is acyclic. For example, the affinity graph in Fig. [1a] is acyclic. We consider this family of affinity-mask sets in Sec. IV.

As mentioned in Sec. I, hierarchical affinity masks have received prior attention [8], [9], [29]. For a given task set $τ$, we call $α_τ$ and $AG(τ)$ hierarchical if and only if, for any $i$ and $j$, $α_i \cap α_j = \emptyset$ or $α_i \subseteq α_j$ or $α_i \supseteq α_j$. Note that hierarchical masks may or may not be acyclic. Note also that the core assignments of any global, clustered, or partitioned scheduler can be specified using masks that are hierarchical. Under global and clustered scheduling, if at least two tasks are allowed to share at least two cores, then such masks will not be acyclic. We consider hierarchical masks in detail in Sec. VII. If we place no restrictions on affinity masks, then $α_τ$ and $AG(τ)$ are called arbitrary. We consider such masks in

¹Our analysis can be extended to support real values at the expense of additional space.
We begin by proving the following lemma, which is true for any arbitrary schedule with bounded tardiness. The lemma statement refers to “uncompleted work.” In a given schedule, the uncompleted work at time $t$ is the total execution time of all jobs released prior to $t$ minus the processing capacity already allocated to those jobs.

**Lemma 1.** If the tardiness of every task in $\tau$ is at most $B$ in some schedule, then at any time instant in that schedule, the amount of uncompleted work is at most $BU + 2\sum_i C_i$.

**Proof.** If tardiness never exceeds $B$, then every job completes within $B$ time units of its deadline, which for a job of task $\tau_i$ is within $B+T_i$ time units of its release. Thus, at any time $t$, all jobs of task $\tau_i$ released prior to time $t-B-T_i$ are completed. During $[t-B-T_i, t)$ task $\tau_i$ may release at most $\lceil \frac{B+T_i}{H_i} \rceil$ jobs. Thus, the amount of uncompleted work due to $\tau_i$ at time $t$ is at most $C_i \lceil \frac{B+T_i}{H_i} \rceil \leq C_i \left( 2 + \frac{B}{H_i} \right) = 2C_i + B \frac{C_i}{H_i} = 2C_i + BU_i$. Summing over all tasks yields $BU + 2\sum_i C_i$. \qed

Returning to Utilization Balance, we have the following.

**Lemma 2.** If $\tau$ is SRT-feasible, then it satisfies Utilization Balance.

**Proof.** Assume, contrary to the statement of the lemma, that a SRT-feasible task set $\tau$ exists that violates Utilization Balance. Then, for some $\tau' \subseteq \tau$,

$$|\alpha_{\tau'}| < U_{\tau'}.$$  

(1)

Consider the following periodic release sequence for $\tau$: each task $\tau_i$ in $\tau$ releases jobs every $T_i$ time units, starting at time 0, and each such job executes for $C_i$ time units. Let $S$ be the schedule with bounded tardiness for this release sequence mentioned in Lemma 1.

By the definition of $\alpha_{\tau'}$, all jobs of all tasks in $\tau'$ are scheduled in $S$ on cores from $\alpha_{\tau'}$. Let $H$ be the hyperperiod of $\tau$. Then, for any integer $k$, the amount of work generated by $\tau'$ over $[0, kH)$ is $\sum_{\tau_i \in \tau'} C_i \cdot kH = kHU_{\tau'} > kH|\alpha_{\tau'}|$, where the last inequality follows from (1). Observe that $kH|\alpha_{\tau'}|$ corresponds to the total available capacity over $[0, kH)$ on cores in $\alpha_{\tau'}$. Thus, the uncompleted work at time $kH$ in $S$ is at least $kH(U_{\tau'} - |\alpha_{\tau'}|)$. This value grows unboundedly with increasing $k$, contradicting Lemma 1. \qed

From results presented later, it will follow that Utilization Balance is a necessary and sufficient condition for SRT-feasibility (and also HRT-feasibility), i.e., Lemma 2 can be strengthened by specifying “if and only if.” When we henceforth wish to emphasize this usage of Utilization Balance, we will refer to it as the UB Test. Unfortunately, applying the UB Test by considering different subsets of tasks can require $\mathcal{O}(2^n)$ time. However, the structure of this test is similar to the famous condition of Hall’s Marriage Theorem [17], the proof of which involves examining maximal “edge matchings” in a graph. Such matchings can be determined by considering the Ford-Fulkerson max-flow algorithm and its correctness proof [14]. This connection to prior work (along with the existence
of polynomial-time algorithms for max flow) motivates us to determine whether max flow can be used to efficiently determine SRT-feasibility.

B. Max-Flow Feasibility Test

To cast checking feasibility as a max-flow problem, we define for any task set \( \tau \) a flow network \( FN(\tau) \) that is obtained from its affinity graph \( AG(\tau) \) via several steps. First, each edge \((\tau_i, \tau_j)\) in \( AG(\tau) \) is viewed as a directed edge from \( \tau_i \) to \( \tau_j \) with capacity \( Z \), where \( Z > m \). Second, a source vertex \( s \) is added along with an edge \((s, \tau_i)\) with capacity \( U_i \) for each vertex \( \tau_i \). Finally, a sink vertex \( t \) is added along with an edge \((\pi_j, t)\) with capacity 1.0 for each vertex \( \pi_j \). Following conventional notation, we let \( f \) denote a flow that is defined with respect to \( FN(\tau) \), with \( f(u, v) \) denoting the flow from vertex \( u \) to vertex \( v \), and we let \( |f| \) denote the value of the flow \( f \) (which equals the total flow from the source \( s \)). (To avoid notational clutter, we do not parameterize \( f \) by \( \tau \).)

**Example 3.** Fig. [3a] shows the flow network corresponding to the affinity graph in Fig. [1b].

**Lemma 3.** If Utilization Balance holds for \( \tau \) and \( f \) is a maximum flow, then \(|f| = U\).

**Proof.** Assuming \( f \) is a maximum flow, by the Max-Flow/Min-Cut Theorem [14], \(|f|\) equals the capacity of a minimal cut. A cut is a partitioning of vertices that places the source and sink in different partitions. The capacity of a cut is simply the sum of the capacities of all edges that traverse the cut. Such an edge is called a crossing edge. For example, Fig. [3b] shows one of the many cuts that can be defined with respect to the flow network in Fig. [3a] (The vertex sets \( V_U \), \( V_W \), and \( V_X \) are discussed later.) This cut has capacity \( U_4 + 2 \).

Let \( C \) be a cut with minimal capacity. If any edge of the form \((\tau_i, \pi_j)\) is a crossing edge, then because its capacity \( Z \) exceeds \( m \) and the capacity of any edge is non-negative, the capacity of \( C \) exceeds \( m \). This cannot be the case if \( C \) is minimal because the cut that places \( s \) and all other vertices in different partitions has capacity \( m \). Thus, every crossing edge is of the form \((s, \tau_i)\) or \((\pi_j, t)\). The cut shown in Fig. [3b] has this property, so the reader may wish to consult it for illustrative purposes hereafter.

Let \( V_W = \{\tau_i \mid (s, \tau_i) \text{ is a crossing edge}\} \), \( V_U = \{\tau_i \mid (s, \tau_i) \text{ is not a crossing edge}\} \), and \( V_X = \{\pi_j \mid (\pi_j, t) \text{ is a crossing edge}\} \). Then, the capacity of the cut \( C \) is \( \sum_{\tau_i \in V_W} U_i + |V_X| \). As long as there are no crossing edges of the form \((\tau_i, \pi_j)\), all edges from tasks in \( V_W \) are directed to \( V_X \). Thus, \( |V_X| \geq |V_W| \). Assuming Utilization Balance holds for \( \tau \), \( |V_X| \geq |\alpha U_V| \). Therefore, the capacity of \( C \) is at least \( \sum_{\tau_i \in V_W} U_i + |V_X| \geq \sum_{\tau_i \in V_W} U_i + |V_X| = \sum_{\tau_i \in \tau} U_i = U \).

To summarize, a minimal cut has capacity at least \( U \). However, the cut that places \( s \) and all other vertices in different partitions has capacity \( U \), so the capacity of any minimal cut is \( U \). Thus, by the Max-Flow/Min-Cut Theorem, \(|f| = U\). □

It will follow from results presented next that showing that \( U \) is a maximum flow is an alternative way to test SRT-feasibility. We refer to this alternative as the MF Test.

In fact, we are going to show that the UB Test and MF Test are each valid tests for both HRT- and SRT-feasibility. We do this by providing reasoning for the remaining links in the proof overview given earlier in Fig. [2]. One of these links involves considering the HRT-feasibility test, denoted APA-Feas(\( \tau, \pi \)), presented by Baruah et al. [7].

**APA-Feas(\( \tau, \pi \)) Test**: declare \( \tau \) to be HRT-feasible if and only if values exist for the variables \( x_{ij} \) satisfying:

\[
\forall i : \sum_{j \in \alpha_i} x_{ij} = 1, \quad \forall j : \sum_{i \in \alpha_j} x_{ij} U_i \leq 1, \quad \text{and} \quad \forall i, j : x_{ij} \geq 0.
\]

**Max-flow linear program.** The APA-Feas(\( \tau, \pi \)) Test can be cast as a linear program (LP). Thus, to make a connection between it and our earlier results, we consider an LP-based implementation of the MF Test, which we refer to as the LP-MF Test:

**LP-MF Test:**

Maximize \( \sum_i f(s, \tau_i) \)

Subject to:

- C1: \( \forall i : 0 \leq f(s, \tau_i) \leq U_i \) \( \{(s, \tau_i) \text{ edges}\} \)
- C2: \( \forall i, j : f(s, \tau_i) \leq Z \) \( \{(\tau_i, \tau_j) \text{ edges}\} \)
- C3: \( \forall j : 0 \leq f(\pi_j, t) \leq 1 \) \( \{(\pi_j, t) \text{ edges}\} \)
- C4: \( \forall i : f(s, \tau_i) = \sum_{j \in \alpha_i} f(\tau_i, \pi_j) \text{ \( \{\tau_i \text{ vertices}\} \)) \)
- C5: \( \forall j : \sum_{i \in \alpha_j} f(\tau_i, \pi_j) = f(\pi_j, t) \text{ \( \{\pi_j \text{ vertices}\} \)) \)

The edge constraints ensure that edge capacities are respected and the vertex constrains ensure that the flow into each non-source/sink vertex matches the flow out of that vertex.

We now show that if \(|f| = U\) holds, then the constraints in the LP above can be simplified, yielding constraints quite
similar to those in the APA-Feas(τ, π) Test. In particular, if $f$ is a maximum flow, then it still lies in the feasibility region of the simplified LP.

**Lemma 4.** If $f$ is a maximum flow and $|f| = U$, then the following conditions hold.

\begin{align*}
\forall i : \sum_{j \in \alpha_i} f(\tau_i, \pi_j) &= U_i \quad (2) \\
\forall j : \sum_{i} f(\tau_i, \pi_j) &\leq 1 \quad (3) \\
\forall i, j, \text{where } j \notin \alpha_i : f(\tau_i, \pi_j) = 0, \quad (4)
\end{align*}

assuming we assign $f(\tau_i, \pi_j)$ to be 0 for $j \notin \alpha_i$ (note that, if $j \notin \alpha_i$, then the edge $(\tau_i, \pi_j)$ is not present in the flow network).

**Proof.** Let $f$ be a maximum flow such that $|f| = U$. Because $f$ is a maximum flow, it satisfies the constraints of the LP of the LP-MF Test. By Constraint C1, $|f| = \sum_i f(s, \tau_i) \leq \sum_i U_i = U$. Because $|f| = U$, by the construction of the flow network, $f(s, \tau_i) = U_i$ holds for each $i$. Thus, by Condition C4, (2) holds. Furthermore, by Conditions C3 and C5, $\sum_{i,j \in \alpha_i} f(\tau_i, \pi_j) = f(\pi_j, t) \leq 1$, so (3) holds, assuming we assign $f(\tau_i, \pi_j)$ to be 0 for $j \notin \alpha_i$ as stated in the lemma. Note that such an assignment trivially satisfies (4). □

**C. HRT- and SRT-Feasibility Equivalence**

The following theorem summarizes the results above.

**Theorem 1.** A task set $\tau$ is SRT-feasible if and only if it is HRT-feasible. Moreover, the UB Test and the MF Test (and its LP counterpart, the LP-MF Test) are each both exact SRT- and HRT-feasibility tests.

**Proof.** By Lemma 2 if $\tau$ is SRT-feasible, then it satisfies Utilization Balance, which by Lemma 3 implies that $|f| = U$ holds, where $f$ is a maximum flow. Thus, by Lemma 4, $f$ satisfies Conditions (2)–(4). Now, defining $x_{ij}$ by $x_{ij} = f(\tau_i, \pi_j)/U_i$, Conditions (2)–(4) imply that all of the conditions of the APA-Feas(τ, π) Test are satisfied, so $\tau$ is HRT-feasible. As noted earlier, HRT-feasibility trivially implies SRT-feasibility. Thus, all links in the chain of reasoning depicted in Fig. 2 have been validated. □

**Remarks.** Given Theorem 1, we will generally use the term “feasible” hereafter without qualifying whether we mean SRT- or HRT-feasibility.

Using a max-flow algorithm from [19] with $\tilde{O}(E \sqrt{V})$ time complexity, where $V$ is the number of vertices and $E$ is the number of edges in the flow network, the MF Test can be performed in $\tilde{O}(mn \sqrt{m+n})$ time, since our flow network satisfies $V = m+n+2$ and $E \leq mn+m+n$. In contrast, the APA-Feas(τ, π) Test requires solving an LP with $mn$ variables and $n+m$ constraints, which requires $Ω((mn+m+n)^{ω+0.5})$ total time in the worst case [19], where $2 < ω < 2.4$ is the matrix multiplication constant [20]. Thus, the MF Test is considerably more efficient than the APA-Feas(τ, π) Test.

---

**IV. ACYCLIC AFFINITIES**

As a stepping stone towards defining algorithm AM-Red, we provide in this section an SRT-optimal scheduler under the restriction that $ω$ is acyclic. For any feasible task set $τ$ that this scheduler must correctly schedule, we fix $f$ to be a maximum flow satisfying Conditions (2)–(4) of Lemma 4. Using this fixed $f$, the algorithm designed here seeks to ensure that each task $τ_i$ receives a long-term processor share on core $π_j$ equal to $f(τ_i, π_j)$.

**Share graph.** Note that $f(τ_i, π_j) = 0$ may hold even when $π_j \in \alpha_i$. In this case, even though task $τ_i$ is allowed to execute on core $π_j$, the share allocation defined above will preclude this from happening. Thus, while the affinity graph $AG(τ)$ includes the edge $(τ_i, π_j)$, this edge can be ignored without affecting schedulability. To reflect this, we define a share graph $SG(τ)$ that is a subgraph of $AG(τ)$. The two are the same except that, in $SG(τ)$, any edge $(τ_i, π_j)$ in $AG(τ)$ for which $f(τ_i, π_j) = 0$ holds in the corresponding flow network $FN(τ)$ is removed. Note that $SG(τ)$ is acyclic if $AG(τ)$ is.

**Example 4.** Consider a task set $τ$ consisting of three tasks, $τ_1$, $τ_2$, and $τ_3$, scheduled on four cores, $π_1$, $π_2$, $π_3$, and $π_4$. If the max-flow values computed for $τ$ are as specified in Fig. 5a, then its share graph is as depicted in Fig. 5b (Fig. 5 has several other insets that are considered later.)

**A. Frames**

To realize the long-term per-task processor shares that we want, we define allocations offline for a time interval called a frame. We denote the allocation function by $F$ and the frame length by $|F|$. At runtime, we use $F$ to perform allocations within each successive time window of length $|F|$. Formally, $F$ is a mapping $F : [0, |F|] \times \{π_1, ..., π_m\} \rightarrow \{0, π_1, ..., π_n\}$. Informally, at each time instant within a window of length $|F|$, $F$ indicates which task is scheduled on each core (if core $π_j$ is idle at time instant $t$, then $F(t, π_j) = 0$).

Let $I_F(τ_i, π_j)$ be the union of all maximal continuous intervals on core $π_j$ allocated to task $τ_i$ by $F$. (Note that we use half-open intervals of the form $[t,F)$.) Then, $F$ is termed valid if the following conditions hold.

\begin{align*}
\forall i,j,j' : I_F(τ_i, π_j) \cap I_F(τ_i, π_j') = \emptyset \quad (5) \\
\forall i,j : |I_F(τ_i, π_j)| = f(τ_i, π_j) \cdot |F| \quad (6)
\end{align*}

(5) implies that $τ_i$ cannot be allocated on different cores simultaneously, while (6) states that task $τ_i$ receives a total per-frame allocation of $f(τ_i, π_j) \cdot |F|$ on core $π_j$.

**Example 5.** Consider the task set $τ$ from Ex. 1 with flow values $f(τ_1, π_1) = 1/4$, $f(τ_2, π_1) = 1/4$, $f(τ_3, π_1) = 1/4$, $f(τ_3, π_2) = 3/8$, $f(τ_4, π_2) = 1/4$ and $f(τ_4, π_3) = 1/4$. A valid frame for $τ$ is depicted in Fig. 4b. Observe that $|F| = 8$ and (for example) $τ_3$’s total allocation on core $π_2$ is $f(τ_3, π_2) \cdot |F| = \frac{3}{8} \cdot 8 = 3$.

---

1$\tilde{O}$ ignores logarithmic factors: $\tilde{O}(g(n)) = O(g(n) \log^k g(n))$ for some natural number $k$. $Ω$ is used similarly in expressing lower bounds.

3We place no restrictions on $|F|$, but its value should be defined and fixed before runtime.
Extended frames. To simplify the problem of defining a valid frame $F$ for $\tau$, we introduce the concept of an extended frame $E$. $E$ is a mapping similar to $F$ but its length is not a priori bounded: $E : [0, \infty) \times \{\pi_1, \ldots, \pi_m\} \to \{0, 2, \ldots, \tau_n\}$. We will show that a valid frame $F$ can be obtained by first constructing $E$ and then “wrapping” the allocations given by $E$ to obtain $F$. First, we need to give validity conditions for $E$. To simplify the problem of defining a valid extended frame $E$, we will first consider an example. Consider the cores $\pi_1, \pi_2, \pi_3$ and tasks $\tau_1, \tau_2, \tau_3$. Assume that the extended frame $E$ can be obtained by first constructing $\pi$ by (7) and (8), and (10). Constructing a valid extended frame. Given Lemma 5 we can focus our attention now on constructing a valid extended frame. To motivate some of the issues that arise in doing so, we first consider an example. Example 7. We can compute a valid extended frame $E$ for the task set in Ex. 5 with $|F| = 1$ as follows. Consider the cores in order, and for each core $\pi_j$, consider the tasks connected by an edge to $\pi_j$ in turn. This ordering is illustrated in Fig. 5c as an “outer” ordering of cores and an “inner” ordering of tasks. Given this ordering, we can apply the simple scheme in Alg. 4 to obtain $E$. First, consider core $\pi_3$ and tasks $\tau_1$ and $\tau_2$ in turn. Allocate them shares of 1/3 and 2/3, respectively, on $\pi_1$, starting from time 0. Now, move on to core $\pi_2$ and consider the tasks $\tau_1$ and $\tau_3$ in turn. Allocate them shares of 1/3 and 2/3, respectively, on $\pi_2$, but this time starting from the end of $\pi_1$’s allocation on $\pi_1$ (so that (11) is not violated). Continue to consider cores and tasks in this manner until all tasks have received their needed share allocations. The resulting extended frame $E$ that is constructed is shown in Fig. 5d.

To show that $F$ is valid, we must show that (5) and (6) hold. By (8), (9), and (10), $L_E(\tau_j)$ has length $\left(\sum_{j \in \alpha_i} f(\tau_i, \pi_j)\right) \cdot |F| = U_i \cdot |F| \leq |F|$, where the first equality follows from (2). Thus, a similar argument as given in the first paragraph of the proof can be applied to show that (5) holds. As for (6), it follows directly from (10).

Ordering cores and tasks. The determination of $E$ worked out easily in Ex. 7 because we conveniently ordered cores and tasks in order to make this happen. Other orderings could be problematic. For example, with the core ordering $\pi_1, \pi_4, \pi_2, \pi_3$, the obtained extended frame $E$ would not be valid because (8) would be violated for $\tau_3$, as illustrated in Fig. 5d. Properly ordering cores is not enough. For example, had we kept the original core ordering but changed the ordering of tasks on core $\pi_2$, placing $\tau_3$ before $\tau_1$, then the obtained extended frame would again violate (8), this time for $\tau_1$, as illustrated in Fig. 5g. These examples show that properly ordering cores and tasks is crucial for Alg. 4 to be correct.

Proper orderings. To correctly apply Alg. 4 in a general way, we define a total order $\preceq$ on cores, and for each core $\pi_j$, a total order $\tau_{j,k}$ on a certain subset of tasks $\tau_{j,k}$. We say that a
holds, so (10) holds. (13) and (14) ensure that, if task \( \tau \) on any interval of size

\[
\pi_1 \quad \frac{1}{2} \quad \frac{2}{3} \quad -
\pi_2 \quad \frac{1}{3} \quad - \quad \frac{2}{3}
\pi_3 \quad \frac{1}{3} \quad - \quad -
\pi_4 \quad - \quad - \quad \frac{1}{3}
\]

\( \forall \pi \) \quad (improper core ordering).

\[ \forall i,j : \pi_i \in \pi_j \text{ if and only if } f(\pi_i, \pi_j) > 0 \] (12)

(13) and (14) ensure that, if task \( \tau \) has allocation intervals placed on different cores, then it can be a non-first task only on the first (by \( \prec \)) of these cores. Thus, Alg. 1 ensures that these allocation intervals are contiguous, so (8) and (11) hold.

Algorithm 2 Proper-Orders Generator

Require: acyclic task graph \( SG(\tau) \)
Ensure: proper core/task order

1: run BFS(\( SG(\tau) \)) \( \Rightarrow \) breadth-first search of \( SG(\tau) \)
   \( \Rightarrow \) Every connected component of \( SG(\tau) \) is searched starting
   with an arbitrary vertex
2: define cores order, \( \prec \), by ordering cores according to their BFS discovery times
3: for \( \pi_j \in \text{cores order} \) do
   4: \( \pi_j \leftarrow \{ \pi_i \mid f(\pi_i, \pi_j) \neq 0 \} \)
   5: define the task order for \( \pi_j \)
   6: if \( \pi_j \) is discovered in BFS by traversing the edge \( (\pi_i, \pi_j) \) then place \( \pi_i \) first
   7: order other tasks in \( \pi \) arbitrarily, after \( \pi_i \) (if it exists)

Generating proper orders. The final issue that remains is actually generating proper cores/tasks orders. For this, we provide Alg. 2. (Note that, if \( SG(\tau) \) is not connected, then we assume that the breadth-first-search routine searches every connected component.)

Theorem 2. If \( SG(\tau) \) is acyclic, then Alg. 2 produces orders that are proper.

Proof. Line 4 of Alg. 2 ensures that (12) holds. In the rest of the proof, we verify the remaining properies, (13) and (14). We assume that all vertices and edges referenced in verifying these properties are part of the same connected component of \( SG(\tau) \).

Assume, to the contrary of (13), that \( \pi_i \) is a non-first task on two distinct cores, \( \pi_j \) and \( \pi_j' \). Then, \( SG(\tau) \) has edges \( (\pi_i, \pi_j) \) and \( (\pi_i, \pi_j') \). Furthermore, \( \pi_j \) and \( \pi_j' \) were discovered before \( \pi_i \). Without loss of generality, assume that \( \pi_j \) was discovered first. Then, there exists a path \( \pi_j \rightarrow \pi_j' \) that does not include \( \pi_i \). Thus, we have a cycle, \( \pi_j \rightarrow \pi_j' \rightarrow \pi_i \rightarrow \pi_j \), which is a contradiction.

Finally, assume to the contrary of (14), that \( \pi_i \) is non-first on core \( \pi_j \), but core \( \pi_j' \) exists such that \( \pi_j' \prec \pi_j \) and \( \pi_j \in \pi_j' \). Because \( \pi_j' \prec \pi_j \), \( \pi_i \) was discovered before \( \pi_j \). Because \( \pi_j \in \pi_j' \), the edge \( (\pi_i, \pi_j) \) exists. Cores are not connected by edges in \( SG(\tau) \), so these facts imply that \( \pi_i \) was discovered before \( \pi_j \). Because \( \pi_i \) was selected as a non-first task on core \( \pi_j \), the edge \( (\pi_i, \pi_j) \) exists. It follows that \( \pi_j \) would have been discovered by traversing that edge, making \( \pi_j \) the first-ordered task on core \( \pi_j \), which is a contradiction.

B. Scheduler

We summarize our results so far by presenting Alg. 3 our algorithm for scheduling task sets with acyclic affinity masks. In referring to a schedule produced by this algorithm, we call a task migrating if it has allocations on multiple cores and fixed otherwise. We present analysis pertaining to this scheduler below, after first providing an example that illustrates how it works.

Example 9. Consider the task set from Ex. 1 with \( f(\pi_i, \pi_j) \) values from Ex. 5 and the (valid) frame \( F \) shown in Fig. 4b.
By assumption, $M$ number of tasks, we therefore have at most one incident edge (if the task set is feasible). It follows that at most $n^2$ it has at most $n$ migrates, optimality, and time complexity.

Frame-based schedulers were first studied years ago [25]. More recently, several frame-based semi-partitioned schedulers have been proposed, [9, 18, 27, 31], but none support affinities. Our frame-based scheduler draws inspiration from one of these [31], but that scheduler is directed at uniform heterogeneous multiprocessors.

C. Analysis

In this section, we analyze Alg. 3 from the perspectives of task migrations, optimality, and time complexity.

Migrations. Let $M$ denote the number of migrating tasks in Alg. 3. The following theorem shows that Alg. 3 limits $M$ in accordance with the idea of semi-partitioned scheduling, where the goal usually is to have $M = O(m)$.

Theorem 3. $M \leq m - 1$.

Proof. By assumption, $SG(\tau_i)$ is acyclic, which implies that it has at most $n + m - 1$ edges. $M$ migrating tasks have at least $2M$ incident edges. Thus, the number of edges incident upon fixed tasks is at most $n + m - 1 - 2M$. Each fixed task has one incident edge (if the task set is feasible). It follows that at most $n + m - 1 - 2M$ tasks are fixed. Because $n$ is the total number of tasks, we therefore have $n \leq M + n + m - 1 - 2M$, implying $M \leq m - 1$.

We now prove several migration-related bounds, all of which are tight, i.e., task sets can be defined for which exactly these bounds hold. In proving these bounds, we let $F$ be the (valid) frame used by Alg 3 and let $deg(\tau_i)$ denote the degree of $\tau_i$ in $SG(\tau)$. When we refer below to an allocation interval for a task $\tau_i$, we mean a maximal continuous interval during which $\tau_i$ is allocated capacity on one core.

Lemma 7. $\tau_i$ has at most $\deg(\tau_i) + 1$ allocation intervals in $F$.

Proof. It is straightforward to show that, in the valid extended frame $E$ that is used to obtain $F$, task $\tau_i$ has at most $\deg(\tau_i)$ allocation intervals. Using [7], [11], it is easy to show that these intervals occupy a continuous time window of length at most $|F|$. Thus, when this interval is “wrapped” (see Lemma 5) to produce $F$, at most one of these intervals is split into two subintervals (e.g., task $\tau_3$ on core $\pi_2$ in Fig. 6). The stated bound follows.

Theorem 4. The overall number of migrations within $F$ is at most $2m - 2$.

Proof. Let $\tau^M$ denote the set of migrating tasks. To compute the overall number of migrations, we consider only these tasks. By Lemma 7 each such task has at most $\deg(\tau_i) + 1$ allocation intervals in $F$. This yields a bound of $\deg(\tau_i)$ for the number of migrations by $\tau_i$ in $F$ because a task’s first allocation interval does not entail a migration. Recall (from the proof of Theorem 3) that $SG(\tau)$ has at most $n + m - 1$ edges. Thus, the total number of migrations within $F$ is at most $\sum_{\tau_i \in \tau^M} \deg(\tau_i) = \text{no. of edges in } SG(\tau) - \sum_{\tau_i \notin \tau^M} \deg(\tau_i) \leq n + m - 1 - (n - M) = m - 1 + M$, which by Theorem 5 is at most $2m - 2$.

Theorem 5. Within any continuous time interval of length $L$, task $\tau_i$ has at most $\lfloor L/|F| \rfloor \cdot \deg(\tau_i)$ migrations, and the overall number of migrations is at most $\lfloor L/|F| \rfloor \cdot (2m - 2)$.

Proof. As discussed in the proof of Lemma 7 when “wrapping” the extended frame $E$ to get $F$, a task $\tau_i$ has at most one allocation interval that is split. If it has zero (e.g., task $\tau_4$ in Fig. 5), then its first and last allocations per frame are for different cores, while if it has one (e.g., task $\tau_3$ on core $\pi_2$ in Fig. 3), they are for the same core. Thus, with zero split allocations, inter-frame migrations can occur, while with one, they cannot. Hence, reasoning as in the proofs of Lemma 7 and Theorem 4 when accounting for both intra- and inter-frame migrations, we have at most $\deg(\tau_i)$ migrations for $\tau_i$ per frame and at most $2m - 2$ per frame overall. Thus, within $L$, $\tau_i$ experiences at most $\lfloor L/|F| \rfloor \cdot \deg(\tau_i)$ migrations, and the overall number of migrations is at most $\lfloor L/|F| \rfloor (2m - 2)$.

Optimality. The next lemma shows that each task receives a long-term processor share equal to its utilization. We use this lemma in showing that Alg. 3 is optimal below.

Lemma 8. Task $\tau_i$ receives a total allocation of at least $U_i \cdot |F|$ during any interval of length $L$.

Proof. During an interval of length $|F|$, $\tau_i$ receives an allocation of $U_i \cdot |F|$. (Note that, if such an interval begins within a frame, then the missing part of that frame is compensated for by the beginning of the next frame—recall that all frames are identical.) During an interval of length $L$, at least $\lfloor L/|F| \rfloor$ complete intervals of length $|F|$ occur.
Theorem 6. Alg. 3 is SRT-optimal and ensures a tardiness bound of $|F|$. Moreover, if $|F|$ divides all periods, then it is HRT-optimal.

Proof. Consider a job $J_{i,s}$ that is released at time $t_r$ and has a deadline at time $t_d$. Let $t$ be the last time instant at or before $t_r$ such that no job of $\tau_i$ released prior to $t$ is unfinished at $t$. Let $k$ be number of jobs of $\tau_i$ released in $[t, t_d)$. Then, because $\tau_i$ has a deadline at $t_d$, we have $t_d - t \geq kT_i$. Thus, by Lemma 8, the total processor allocation received by $\tau_i$ during $[t, t_d+|F|)$ is at least

$$U_i \cdot |F| \cdot \left[ \frac{t_d + |F| - t}{|F|} \right] \geq U_i \cdot |F| \cdot \left( \frac{k \cdot T_i}{|F|} + 1 \right) \geq U_i \cdot |F| \cdot \left( \frac{k \cdot T_i}{|F|} \right) = k \cdot C_i.$$ 

This implies that $J_{i,s}$ completes by time $t_d + |F|$, i.e., Alg. 3 is SRT-optimal and ensures a tardiness bound of $|F|$.

If $|F|$ divides all task periods, then similar reasoning can be applied, but this time with respect to the interval $[t, t_d)$. Because $|F|$ divides $T_i$, by Lemma 8 the total processor allocation received by $\tau_i$ during this interval is at least

$$U_i \cdot |F| \cdot \left[ \frac{t_d - t}{|F|} \right] \geq U_i \cdot |F| \cdot \left( \frac{k \cdot T_i}{|F|} \right) = U_i \cdot |F| \cdot \left( \frac{k \cdot T_i}{|F|} \right) = k \cdot C_i.$$ 

This implies that $J_{i,s}$ completes with zero tardiness and that Alg. 3 is HRT-optimal.

Time complexity. It is straightforward to show that Algs. 1 and 2 each can be implemented in $O(m + n)$ time (under the assumption of correct input). In contrast, the most efficient known max-flow algorithms require super-linear time. Thus, the time complexity required by Alg. 3 to find a valid frame is dominated by that of the max-flow algorithm that is used. Lee et al. [19] have presented a max-flow algorithm that has time complexity $O(mn \sqrt{m + n})$ for an arbitrary $AG(\tau)$ and $O((m + n)^{3/2})$ in our setting (the number of edges in an acyclic $AG(\tau)$ is limited by $(m + n - 1)$), giving us the following.

Theorem 7. For any feasible task set $\tau$ with acyclic $\alpha_\tau$, Alg. 3 can produce a valid frame $F$ in $O(mn \sqrt{m + n})$ time.

Bonifaci et al. [9] claim that semi-partitioned and hierarchical scheduling with affinity masks is NP-hard to approximate.

Their work does not contradict Theorem 7 because it is directed at a completely different context, namely, one-shot jobs and makespan minimization.

V. ARBITRARY AFFINITY MASKS

In this section, we show that arbitrary affinities can be dealt with by eliminating any cycles that may exist in $SG(\tau)$. First, we show how to remove a single cycle from $SG(\tau)$.

Cycle removal procedure. Because $SG(\tau)$ is bipartite, any cycle is of the form $\tau_i \rightarrow \pi_{j_1} \rightarrow \tau_{i_2} \rightarrow \pi_{j_2} \rightarrow \ldots \rightarrow \pi_{j_m} \rightarrow \tau_i$. Let $f_m = \min(f(\pi_{i_1}, \pi_{j_1}), f(\pi_{i_2}, \pi_{j_2}), \ldots, f(\tau_{i_3}, \pi_{j_m}))$, i.e., $f_m$ is the minimal $f$ value of any task-to-core edge in this cycle. Then, we can eliminate the cycle by decreasing the $f$ value of each core-to-task edge by $f_m$ and by increasing the $f$ value of each task-to-core edge by $f_m$. This eliminates all cycle edges with $f$ values of $f_m$. As a result, the cycle is eliminated because at least one of its edges is removed (recall that all edges have non-zero weights in $SG(\tau)$). Note that this procedure does not change $\sum_{\pi_j} f(\tau_i, \pi_j)$ for any $\pi_j$ or $\sum_{\tau_i} f(\tau_i, \pi_j)$ for any $\tau_i$. Thus, all conditions of Lemma 4 hold for $f$.

Example 10. Applying this cycle-removal procedure to the share graph in Fig. 7a, we have $f_m = 0.1$, and the share graph in Fig. 7b results.

Now that we have a procedure for eliminating cycles, we need a means for finding them. A breadth-first-search routine can do this, as seen in Alg. 4. It is easy to see that this algorithm produces an acyclic share graph: it does not add any new edges, so it cannot create any new cycles; also, any cycle initially in $SG(\tau)$ will be found by BFS and removed.

To determine the time complexity of Alg. 4 note that each invocation of the BFS routine requires $O(E)$ time, where $E$
Algorithm 4 Affinity-Reduction Algorithm

Require: \( f \)
1: construct \( SG(\tau) \)
2: for \( \tau_i \in \tau \) do
3: \[ \text{while true do} \quad \triangleright \text{process connected comp. (cc) of } SG(\tau) \]
4: \[ \text{Run BFS(\tau_i)} \]
5: \[ \text{if BFS found a cycle in } SG(\tau_i) \text{ then} \]
6: \[ \text{remove edge(s) from the cycle using the procedure} \]
7: \[ \text{described earlier and illustrated in Ex. 10} \]
8: \[ \text{break while} \quad \triangleright \text{no more cycles in the cc of } SG(\tau) \]
9: return obtained \( f \)

Algorithm 5 AM-Red Scheduler

Require: \( \tau, \_frame\_len \)
1: \[ \text{function FeasibilityCheck(\tau)} \quad \triangleright \text{Sec. III-B} \]
2: \[ \text{construct flow network } FN(\tau) \]
3: \[ \text{compute max flow } f \text{ with respect to } FN(\tau) \]
4: \[ \text{if } |f| = U \text{ then return } f \quad \triangleright \text{MF Test} \]
5: \[ \text{else return } \perp \quad \triangleright \tau \text{ is infeasible} \]
6: \[ \text{function GenerateFrame(\tau, _frame_len, f)} \quad \triangleright \text{Sec. IV-A} \]
7: \[ \text{run Alg. 3 to get proper core/task orders} \]
8: \[ \text{run Alg. 4 to build a valid extended frame } E \]
9: \[ \text{apply the transformation of Lemma 3 to } E \]
10: \[ \text{to obtain a valid frame } F \text{ with } |F| = \text{frame\_len} \]
11: \[ \text{return } F \]
12: \[ \text{function GetFlow(\tau)} \]
13: \[ f \leftarrow \text{FeasibilityCheck(\tau)} \]
14: \[ \text{run Alg. 4 to obtain acyclic } f \text{ values} \]
15: \[ \text{return } f \]
16: \[ \text{function Scheduler(\tau, _frame_len)} \]
17: \[ f \leftarrow \text{GetFlow(\tau)} \]
18: \[ \text{if } f \neq \perp \text{ then} \]
19: \[ F \leftarrow \text{GenerateFrame(\tau, _frame_len, f)} \]
20: \[ \text{repeat the allocations in } F \text{ every } |F| \text{ time units, letting} \]
21: \[ \text{the jobs of each task } \tau \text{ execute within the allocation} \]
22: \[ \text{intervals for } \tau \text{ in release-time order} \]

is the number of edges in the initial graph \( SG(\tau) \). In our
context, \( E \) is upper bounded by the number of edges in \( AG(\tau) \),
which is \( O(mn) \). The BFS routine is invoked \( O(E + n) \) times,
because each invocation removes at least one edge or moves
to a new task. The edge-removal procedure itself (illustrated in Ex. 10)
requires \( O(m) \) time. From this discussion, we have the following theorem.

Theorem 8. Alg. 4 transforms \( SG(\tau) \) into acyclic graph. Its
time complexity is \( O(m^2n^2) \) generally, and \( O(n^2) \) if the number
of edges in \( AG(\tau) \) is linear.

Algorithm AM-Red. We are finally in a position to present the
main contribution of this paper, algorithm AM-Red (Alg. 5). It
is obtained by applying the various algorithms presented in this
paper as building blocks in the expected way. The following theorem combines the results of Theorems 3, 4, 5, 6, 7, and 8.

Theorem 9. For any feasible task set \( \tau \), AM-Red produces a
valid frame \( F \) in \( O(m^2n^2) \) time; it ensures that at most \( m - 1 \)
tasks migrate and that at most \( 2m - 2 \) migrations occur per
frame; it is SRT-optimal with a tardiness bound of \( |F| \); if \( F \)
divides all periods, then it is also HRT-optimal.

VI. HIERARCHICAL MASKS

In this section, we show that the relatively high time complexity of the MF Test and affinity reduction (Alg. 4), which dominates the time complexity of AM-Red (Alg. 5), can be avoided if \( \alpha_\tau \) is hierarchical. To facilitate showing this, we assume that tasks are indexed such that \( i \leq j \Rightarrow |\alpha_i| \leq |\alpha_j| \).
We call this ordering canonical order. For now, we assume this
ordering is initially provided. Later we consider the time complexity
to obtain it if not initially provided.

We now establish a simpler feasibility test when \( \alpha_\tau \) is hierarchi-
cal. To facilitate our description of this test, we introduce some
new terminology. We say that task \( \tau_i \) is nested within task \( \tau_j \)
if and only if \( i \leq j \) and \( \alpha_i \subseteq \alpha_j \). It is easy to see that
the “nested within” relation is transitive. We denote the set of
tasks nested within \( \tau_i \) as \( N_i \) (note that \( \tau_i \in N_i \)). Observe that,
if \( \tau_i \) is nested within \( \tau_j \), then \( N_i \subseteq N_j \) by transitivity. For any
task \( \tau_i \), we define its utilization closure as \( U^*_i = \sum_{\tau_j \in N_i} U_j \).
We call a task \( \tau_i \) maximal if it is not nested within any other
task \( \tau_j \) with the same affinity mask, where \( j \geq i \). For any task
\( \tau_i \), we let \( A_i \) denote the set of tasks with the same affinity
mask (note that \( \tau_i \in A_i \)); we say that these tasks agree with
\( \tau_i \). Note that the last task in \( A_i \) (in canonical order) is maximal.
For \( \tau' \subseteq \tau \), we let \( X(\tau') \) denote the set of all maximal tasks
in \( \bigcup_{\tau_i \in \tau'} A_i \), and we let \( X(\tau') \) denote those tasks in \( X(\tau') \)
at the “top” of the nesting hierarchy, i.e., \( X(\tau') = \{ \tau_i : \tau_i \in
X(\tau') \land \tau_i \text{ is not nested within any other task in } X(\tau') \} \).
Note that distinct tasks in \( X(\tau') \) have disjoint masks.

Example 11. Consider task set \( \tau \) from Ex. 2. Its affinity graph
is shown in Fig. 11b. Consider the canonical order \( \tau_1, \tau_4, \tau_2, \tau_3 \).
(When reasoning abstractly, we assume this ordering is con-
sistent with task indices, as noted above.) Then, \( N_1 = \{ \tau_1 \},\]
\( N_2 = \{ \tau_1, \tau_2 \}, N_3 = \{ \tau_1, \tau_2, \tau_3 \}, \) and \( N_4 = \{ \tau_4 \} \).
Also, \( A_1 = \{ \tau_1 \}, A_2 = A_3 = \{ \tau_2, \tau_3 \}, \) and \( A_4 = \{ \tau_4 \} \). Tasks \( \tau_1, \tau_3, \) and \( \tau_4 \) are maximal, but task \( \tau_2 \) is not since it is nested
within \( \tau_3 \). For \( \tau' = \{ \tau_1, \tau_2, \tau_3 \} \), \( X(\tau') = \{ \tau_1, \tau_2, \tau_3 \} \) and
\( X(\tau') = \{ \tau_3 \} \). Note that \( \tau_3 \) and \( \tau_4 \) have disjoint masks.

Lemma 9. For any \( \tau_i \in \tau' \), \( \tau_i \) is nested within some task in
\( X(\tau') \) and also \( X(\tau') \).

Proof. Any task in \( X(\tau') \) is nested within some task in \( X(\tau') \),
so we can limit attention to \( X(\tau') \). If \( \tau_i \in \tau' \) and \( \tau_i \) itself is
not maximal, then it is nested within another task \( \tau_j \) with the
same mask that is maximal. Because \( \tau_i \in \tau' \), \( \tau_j \in X(\tau') \).

The simplified feasibility condition we require is as follows.

Nest ed Balance: For any maximal task \( \tau_i : U^*_i \leq |\alpha_i| \).

Lemma 10. For any hierarchical \( \alpha_\tau \), Utilization Balance (UB)
and Nested Balance (NB) are equivalent.

Proof. It is easy to show UB \( \Rightarrow \) NB: if UB holds, then by
considering \( \tau' = N_i \) in that condition, NB easily follows. In
the rest of the proof, we focus on showing NB \( \Rightarrow \) UB.

Consider any \( \tau' \subseteq \tau \) and any task \( \tau_i \in \tau' \). By Lemma 9, \( \tau_i \)
is nested within some task in \( X(\tau') \). Thus, \( \tau' \subseteq \bigcup_{\tau_j \in X(\tau')} N_j \),
which implies \( U_{\tau'} \leq \sum_{\tau_j \in X(\tau')} U_j^* \). By NB, \( \sum_{\tau_j \in X(\tau')} U_j^* \leq \)
Balance is violated in this case as well. \( \tau \) with the same mask as is infeasible. Otherwise, there exists a maximal task \( \tau \) Balance is violated (and hence, by Theorem 1 and Lemma 10, this implies that the total number of iterations of Alg. 6 is at

\[
\sum_{\tau_j \in \hat{X}(\tau')} |\alpha_j|.
\]

As observed earlier, all tasks from \( \hat{X}(\tau') \) have disjoint masks. Moreover, the cores included in these masks are exactly the same as those included in \( \alpha_\tau \). Hence, \( \sum_{\tau_j \in \hat{X}(\tau')} |\alpha_j| = |\alpha_\tau| \).

Having established a simpler feasibility condition, it remains to show it can be efficiently computed. Alg. 6 does this while also returning all needed non-zero \( f(\tau_i, \pi_j) \) values. On each loop iteration (considering each task \( \tau_i \) in turn), the algorithm fills core \( \pi_j \) fully or fully allocates to \( \tau_i \) its utilization \( U_i \).

Analysis. We now show that Alg. 6 is correct.

**Theorem 10.** Alg. 6 returns \( f(\tau_i, \pi_j) \) values satisfying (27) if and only if \( \tau \) satisfies Nested Balance.

**Proof.** Establishing the algorithm’s correctness when it does not return \( \bot \) for \( \tau \) is straightforward, so we focus on the other possibility, i.e., it returns \( \bot \) when considering some task \( \tau_i \) in \( \tau \). Because tasks are processed in canonical order, by the time \( \tau_i \) is considered, all tasks from \( N_i / \{ \tau_i \} \) have already been dealt with and no task with a larger mask has yet been considered. Thus, the cores in \( \alpha_i \) could only have been allocated to tasks in \( N_i \). If we cannot allocate \( \tau_i \), then

\[
U^*_i = \sum_{\tau_j \in \hat{X}(\tau_i)} U_j > |\alpha_i|.
\]

If \( \tau_i \) is maximal, then Nested Balance is violated (and hence, by Theorem 1 and Lemma 10 \( \tau \) is infeasible). Otherwise, there exists a maximal task \( \tau_k \) with the same mask as \( \tau_i \) but ordered after \( \tau_i \). In this case, \( N_i \subset N_k \), so

\[
U^*_k > \sum_{\tau_j \in \hat{X}(\tau_k)} U_j > |\alpha_k| = |\alpha_i|.
\]

Therefore, Nested Balance is violated in this case as well.

**Theorem 11.** Alg. 6 completes in \( O(m + n) \) time. Thus, if \( \alpha_\tau \) is hierarchical, tasks are indexed in canonical order, and Alg. 6 is used in place of the GetFlow function in AM-Red, then AM-Red produces a valid frame in \( O(m + n) \) time.

**Proof.** During each while-loop iteration, either some core \( \pi_j \) becomes fully allocated (\( \text{cap}[j] \) becomes one), or the current task becomes fully allocated (\( r \) becomes zero), or \( \bot \) is returned. Each of these possibilities may happen only once. This implies that the total number of iterations of Alg. 6 is at most \( m + n \).

If \( \alpha_\tau \) is hierarchical, then only \( O(m) \) unique affinity masks may exist [26 Theorem 3.5]. Thus, only \( O(m + n) \) space is required to provide canonically ordered tasks as input, as each task merely requires a pointer to one of the \( O(m) \) masks that may exist. If canonical order cannot be pre-assumed, then tasks must be sorted so they are so ordered. This can be done in \( O(m \log m + n) \) time: the masks themselves can be sorted in \( O(m \log m) \) time, and all per-task mask pointers can be updated in \( O(n) \) time. If one takes the core count \( m \) to be a constant (which is a reasonable assumption), then the masks can be sorted in \( O(1) \) time and the total time complexity required to put tasks into canonical order is only \( O(n) \).

Example task sets can easily be constructed that have \( \Omega(m) \) distinct hierarchical masks. Because any feasibility test must consider these masks and all tasks, it follows that the \( O(m + n) \) time complexity shown above is asymptotically optimal.

VII. EXPERIMENTAL EVALUATION

This paper is mostly directed at the theoretical aspects of scheduling with affinity masks. In this section, we experimentally evaluate the culmination of this theory, AM-Red, on the basis of overheads and tardiness.

Relevant overheads can be placed into three groups: scheduling overhead (due to invocations of the scheduler), other OS-related overheads (interrupt handling, context switches, etc.), and cache-related preemption and migration delays (CPCMs) (caused by cache-affinity loss due to preemptions and migrations). To assess scheduling and OS overheads, we implemented AM-Red in LITMUSRT [1] and measured these overheads on a 24-core Intel Xeon system. The results we obtained are summarized in Fig. 8. Given the small magnitude of these overheads, they can be easily factored into task execution times when applying AM-Red.

CPCMs can be much larger [10 p. 325] and hinge on task-specific characteristics pertaining to memory usage that would require a more involved study than is possible to fully consider given space constraints. In AM-Red, such overheads are heavily tied to the frequency of task migrations, so to provide a sense of the impact of CPCMs, we conducted experiments in which migration frequency was assessed.

While it would be desirable to compare migration frequency under AM-Red to a range of other algorithms, there is a dearth of prior algorithms capable of handling affinity masks to which to compare. Given this, we compared AM-Red to HPA-EDF, the scheduler proposed in [8]. In this comparison, we were forced to limit attention to hierarchical masks, as HPA-EDF requires this. Because migration frequency hinges only on algorithmic properties, we based our comparison on computed schedules, rather than actual scheduler implementations. As \( |F| \) is tunable parameter, we considered three different ways of choosing it, resulting in three AM-Red variants: AM-Red-min,

---

**Algorithm 6: Nested Balance Test**

**Require:** \( \tau \) (in canonical order)

1: \( \text{cap}[j] = 0 \) \hspace{1em} \( \triangleright \) capacity used on each core, bounded by one
2: for \( \tau_i \in \tau \) do
3: \( r \leftarrow U_i \) \hspace{1em} \( \triangleright \) \( r \) is remaining unallocated utilization of \( \tau_i \)
4: \( \text{while} \ \pi_j \text{ from } \alpha_\tau \text{ with } \text{cap}[j] < 1 \text{ exists and } r > 0 \) do
5: \( f(\tau_i, \pi_j) \leftarrow \min(1 - \text{cap}[j], r) \)
6: \( r \leftarrow r - f(\tau_i, \pi_j) \)
7: \( \text{cap}[j] \leftarrow \text{cap}[j] + f(\tau_i, \pi_j) \)
8: if \( r > 0 \) then return \( \bot \) \hspace{1em} \( \triangleright \) Nested Balance is violated
9: else any non-defined \( f(\tau_i, \pi_j) \) value is considered to be 0

---

![Fig. 8: Scheduling and other OS overheads for AM-Red. Various percentiles are given, as well as averages and medians, as computed over all collected overhead values in each category.](image-url)
for which \(|F| = \min\{T_i\}; \text{ AM-Red-avg, for which } |F| = \text{average}\{T_i\}; \text{ and AM-Red-max, for which } |F| = \max\{T_i\}.\)

**Input-data generation.** To assess migration frequency, we randomly generated sporadic task sets for a 16-core platform (so \(m = 16\)), with task periods selected from the range [40ms, 100ms]. Due to space limitations, we consider only two categories of tasks here: light tasks with \(U_i \in (0.0, 0.3)\), and heavy tasks with \(U_i \in [0.7, 1.0)\). We generated (hierarchical) masks independently of tasks using a process that produced masks for which the number of nesting levels ranged from approximately \(\log m\) to \(m - 1\) (the maximum possible level). We discarded any non-feasible task sets that were generated.

**Migrations.** We recorded, for both schedulers, the number of task migrations over an interval of length \(100s\) as a function of relative system utilization. We concluded by reminding the reader that HPA-EDF is SRT-optimal, with a tardiness bound of \(|F|\), and that it is HRT-optimal if \(|F|\) divides the smallest task period. We also presented analysis concerning task-migration frequency and time complexity. In the special case of hierarchical masks, we showed that AM-Red can be refined to find a valid frame in \(O(m + n)\) time, which is asymptotically optimal.

In other work that we omit due to space constraints, we have shown that the time complexity for frame construction can be reduced in other special cases. For example, for acyclic graphs, it can be reduced to \(O(m + n)\), which again is asymptotically optimal, using techniques similar to those discussed in Sec. [VI].

**VIII. CONCLUSION**

We have presented AM-Red, the first (non-clairvoyant) optimal scheduler for implicit-deadline sporadic task sets assuming arbitrary processor affinity masks. We showed that AM-Red is SRT-optimal, with a tardiness bound of \(|F|\), and that it is HRT-optimal if \(|F|\) divides the smallest task period. We also presented analysis concerning task-migration frequency and time complexity. In the special case of hierarchical masks, we showed that AM-Red can be refined to find a valid frame in \(O(m + n)\) time, which is asymptotically optimal.

If masks are restricted in length, then it can also be reduced due to the internal structure of the affinity graph.

In future work, we intend to adapt AM-Red for heterogeneous multiprocessors, which are becoming more common in practice. Also, although the number of migrating tasks under AM-Red is generally optimal (i.e., task sets exist for which \(m - 1\) migrating tasks are fundamental, matching the bound in Theorem 3), we wish to find a way to reduce the number of migrating tasks to within a constant factor of that optically required for each specific task set under consideration. Finally, while our focus in this paper has been semi-partitioned scheduling, global scheduling warrants consideration.
ACKNOWLEDGMENT

Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.

REFERENCES

user_guide%2FTopic%2FHow_to_Thread_affinity.html
%20SOURCES=DOWNLOAD