Co-Located Parallel Scheduling of Threads to Optimize Cache Sharing

Corey Tessler[†], Prashant Modekurthy[†], Nathan Fisher[§], Abusayeed Saifullah[§], Alleyn Murphy[†]
[†]University of Nevada, Las Vegas, [§]Wayne State University

***Abstract*—For hard-real time systems, cache memory increases execution time variability, increasing the complexity of timing analysis. As such, cache memory is often treated exclusively as a detractor to schedulability. Cache-aware co-located scheduling aims to improve schedulability by carefully scheduling threads to share cached values. Cache sharing between threads potentially reduces task execution times and increases schedulability with fewer resources. Antithetically, co-located scheduling may reduce parallelism, decreasing efficiency. Thus, identifying the optimal set of threads to co-locate that minimizes the resources required while ensuring timing constraints is a complex challenge. This work establishes optimal co-location as NP-Hard in the strong sense. It offers an approximation method for the co-located scheduling of Fork-Join tasks named 3-PARM-HD. The approximation has a 3-factor guarantee and a resource augmentation bound of 3. The simulated evaluation shows 3-PARM-HD increases schedulability compared to an optimal intractable algorithm (without co-location) scheduling 28% more tasks with 30% fewer cores. Simulated results show 3-PARM-HD outperforms a 2-factor approximation for traditional makespan, scheduling 39% more tasks with 44% fewer cores. An experimental RISC-V evaluation running on a QEMU platform confirms the benefits of 3-PARM-HD, scheduling and executing tasks deemed unschedulable by a 2-factor makespan approximation without co-location.**

I. Introduction

Computational demands found in today's safety-critical systems exceed the capacity of a single processor. Multi-processor systems with parallel scheduling algorithms are employed to meet the demands of systems performing autonomous driving, computer vision, object detection, and other complex tasks [39]. For these systems to operate safely their timely operation must be guaranteed.

Ensuring the temporal guarantees required of safety-critical systems is the responsibility of *schedulability analysis*. The schedulability of parallel tasks has been well studied [6]–[8], [10], [16], [17], [19], [24]–[26], [28], [29], [32], [33], [49], [53], [57], [59], [61], [64], [72], [73], [78]–[80]. A persistent challenge for schedulability analysis is memory contention [20], [35].

As a component of the memory system, instruction caches contribute to contention. The variability cache contention introduces to task execution times is typically perceived negatively [5], [54], [77], exclusively increasing execution times. An alternative positive perspective is taken in [70], whereby threads of tasks are scheduled in a cache-aware manner to reduce variability and total execution times. This *cache-aware co-located scheduling* of parallel tasks has the potential to reduce task execution times, increase efficiency, and ensure safety with fewer resources for safety-critical systems.

For parallel directed acyclic graph (DAG) tasks, the distinct execution of threads upon distinct cores may be joined into a single execution request upon a single core [70]. Execution requests are joined by the `BUNDLE` thread-level scheduling algorithm [67], reducing the aggregate execution time of all threads. Doing so reduces the number of cores required to schedule high utilization federated DAG tasks. Joining executions in this way is referred to as *co-location*. Unfortunately, not all threads may be co-located while guaranteeing the timely and safe completion of parallel tasks. This creates a decision problem, where finding the subset of threads to co-locate in order to reduce the number of required cores must be balanced with the timely execution of a task.

The problem of optimal co-location for DAG tasks was introduced in [70], where it was addressed by sub-optimal heuristics. That work did not show the complexity class nor provide a guaranteed approximation. Herein, optimal co-location is proven to be NP-Hard in the strong sense. By establishing the intractability of co-location, research into an exact, tractable algorithm is halted. In place of an exact algorithm, this work provides the *first* guaranteed approximation methods with proven augmentation bounds for Fork-Join parallel tasks. Fork-Join tasks may be seen as restricted forms of DAG tasks. The major contributions of this work are:

1) Proof of strong NP-hardness of optimal co-location.
2) A 3-factor approximation algorithm for minimizing the makespan of a Fork-Join task with cache-aware scheduling with a resource augmentation bound of 3.
3) A simulated evaluation of the approximate and exact methods that may be freely extended and repurposed [66].
4) An empirical Fork-Join scheduling algorithm evaluation for RISC-V [66] utilizing the QEMU platform.

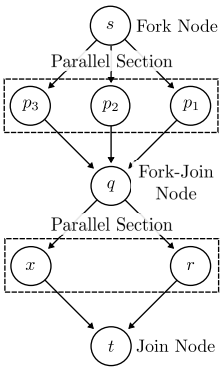
To convey these contributions, the presentation is divided into sections. Section II presents the Fork-Join task model. Section III provides the necessary background. Sections IV and V present the optimal co-location problem and prove its complexity. Two 3-factor approximation methods are proposed in Section VI. Section VII and VIII verify the benefits of co-location through simulations and experimentation. Related work appears in Section IX. Section X concludes.

II. Fork-Join Parallel Task Model

Under the Fork-Join model [36] the individual executable objects of a parallel task are represented as nodes. A Fork-Join task is a series of fork nodes, parallel sections, and join nodes.

A *fork node* has one or more outgoing edges to immediate successor nodes, these successors comprise a parallel section. All nodes of a parallel section have one outgoing edge to a shared immediate successor, a *join node*. A fork node may also be a join node.

Figure 1 illustrates the relationship between an OpenMP [52] fragment and a Fork-Join task graph. From the code fragment, the functions `s()`, `q()`, and `t()` are represented as fork, fork-join, and join nodes within the graph of the task. There are two parallel sections, the first contains three nodes corresponding to three threads executing the function `p()` concurrently. The second parallel section contains two nodes corresponding to two threads executing the functions `r()` and `x()` concurrently. The general Fork-Join [36] model permits embedded parallel sections, e.g. p_1 may represent a fork node, parallel section, and join node. Herein, embedded parallel sections are prohibited.




```
s(); /* fork node*/
#pragma omp for
for (i=1; i<=3; i++) {
  p(i);
}

q(); /* fork-join node */

#pragma omp parallel sections {
  #pragma omp section
  { r(); }
  #pragma omp section
  { x(); }
}
t(); /* join node */
```

Fig. 1: Fork-Join Task and OpenMP Code Fragment

A Fork-Join task τ is represented by a tuple $\tau = (T, D, G)$ where T is the minimum inter-arrival time, D is the relative deadline, and G is the graph of the task. A task's graph $G = (V, E)$ is composed of nodes V and edges E. A node $v \in V$ represents a thread executing in isolation upon a single core and its execution is bounded by its worst-case execution time (WCET) C_v. An edge $(u, v) \in E$ expresses an execution dependency between $u, v \in V$. A node v is ready to execute only when all predecessors ($\{u \mid (u, v) \in E\}$) have terminated. Consequently, only nodes in a single parallel section may execute in parallel.

Fork and join nodes are referred to as *sequential nodes*, the set of sequential nodes is denoted $S \subset V$. Without loss of generality, there are $|S| - 1$ parallel sections.The set of parallel sections is denoted by $P = \{P_1, P_2, ..., P_{|S|-1}\}$ and ordered by increasing distance from the source node. Nodes within a parallel section are referred to as *parallel nodes*.

A task τ generates an infinite number of jobs, arriving at least T time units apart. All jobs of τ must complete before the next job is released (i.e. $D = T$). On a system with $\mathbb{M}$ identical cores, utilizing a non-preemptive, parallel scheduling algorithm, the following definitions of makespan and schedulability apply.

Definition 1 (Makespan of P_i given m Cores). For a parallel section $P_i \in P$, the makespan λ_i of P_i is an upper bound on the amount of time required to execute all threads of all nodes of P_i upon m cores.

Definition 2 (Makespan of G given m Cores). The makespan Λ of $G = (V, E)$ is an upper bound on the amount of time required to execute all sequential nodes and parallel sections upon m cores:

$$\Lambda = \left(\sum_{s \in S} C_s\right) + \left(\sum_{P_i \in P} \lambda_i\right) \tag{1}$$

Definition 3 (Schedulability of G). $G = (V, E)$ is schedulable if the makespan of G is less than or equal to the relative deadline of the task: $\Lambda \leq D$.

Observation 1 (Composition of Λ). Each parallel section contributes independently to the makespan of G by Equation 1.

Observation 2 (Minimum Λ). By Observation 1, minimizing the makespans of the individual parallel sections minimizes the makespan of the task.

III. Background

For parallel hard real-time tasks executing upon a single processor, the `BUNDLE` [67] scheduling algorithm and analysis techniques integrate cache analysis with thread-level scheduling decisions. Cache analysis depends on the *inter-thread cache benefit* (ITCB), which is defined as the reduction in execution time of one thread due, exclusively, to the placement of values in the cache by another thread. The analysis serves as input to the `BUNDLE` scheduling algorithm. The analysis segments threads into *conflict free regions*: sets of instructions that do not evict one another when executed. The `BUNDLE` scheduling algorithm selects one conflict free region as *active*. Threads are associated with regions and only threads of the active region are permitted to execute. Once a thread is selected to execute, it does so non-preemptively until it leaves the active region and then it is blocked. Thus, threads of the active region place values into the cache that cannot be evicted and will be shared by subsequent threads; producing the ITCB. Scheduling repeatedly selects an active conflict free region until all threads terminate. To integrate the ITCB into the WCET of a task, the benefit must be quantifiable. `BUNDLE` analysis and scheduling quantifies and guarantees the ITCB for parallel tasks, reducing their total WCET.

For each task, `BUNDLE` analysis produces a WCET function $c(z) : \mathbb{N} \to \mathbb{R}$, where z is the number of threads released per parallel job. The function returns the WCET of z threads of the same parallel job scheduled upon one processor by `BUNDLE`. By incorporating the ITCB into WCET functions, they become strictly increasing and concave [69] obeying Equation 2.

$$\forall z \in \mathbb{N}, c(z) - c(z-1) \geq c(z+1) - c(z) \tag{2}$$

Due to their concavity, a `BUNDLE` WCET function may be upper bounded by a linear function in the form of Equation 3 taken from [69]. Throughout this work, linear bounding functions take the place of `BUNDLE` WCET functions.

Definition 4 (Bounding Function of BUNDLE WCET). A BUNDLE WCET function may be upper bounded by a linear function $c(z) : \mathbb{N} \to \mathbb{R}$, with $\gamma \in [0, 1]$, of the form:

$$c(z) = \beta + (z - 1)(\gamma\beta) \tag{3}$$

Additional terminology is ascribed to BUNDLE bounding functions and threads. The β and γ terms are referred to as the *base* and *incremental* costs. The first thread with cost β is referred to as a *heavy weight* thread, subsequent threads which increase $c(z)$ by $\gamma\beta$ for each thread are referred to as *light weight* threads. BUNDLE analysis provides the base and incremental costs. Alternatively, they may be determined from the WCET functions, where $\beta = c(1)$ and $\gamma = \frac{c(2)-c(1)}{c(1)}$.

Due to the concavity of the bounding function, combining distinct thread execution requests of the same task strictly decreases the total WCET for the task. This is referred to as the joined request bound property:

Property 1 (Joined Request Bound). Following from Equation 2, given $n, k \in \mathbb{N}$ distinct threads for the same parallel task, the WCET contribution $c(n) + c(k)$ is greater than or equal to the WCET of their combined execution $c(n + k)$, i.e. $c(n + k) \leq c(n) + c(k)$.

BUNDLE analysis produces WCET functions for executable *objects*. An object is a set of instructions. An object may be the complete set of instructions for a logical task, e.g. all instructions reachable from an entry point. Or an object may be a subset of instructions, such as the body of a parallel `for` loop. A *thread* is the execution of an object. Multiple threads may execute in parallel over the same object.

BUNDLE analysis is (currently) limited to level one direct-mapped instruction caches. BUNDLE has been shown [67]–[70] to reduce the WCET and run-time of parallel tasks. In terms of single core performance, BUNDLE reduces WCET times linearly with respect to the number of threads [67]; under the assumption of a block reload time of 10 cycles, a constant cycle per instruction of 1 cycle, and a thread level context-switch cost of 3% of the total execution time of one thread in isolation. In terms of multi-core performance, federeated DAG tasks have their dedicated cores reduced by 25% to 50% [70] on a proof-of-concept distributed parallel processing platform consisting of ARM Cortex A53 processors.

Herein BUNDLE analysis is an opaque process that generates a WCET function and bounding function per object. The bounding functions guide the complexity analysis and co-location algorithms. As such, any BUNDLE analysis improvements (hierarchical caches, data caches, etc.) would increase the benefits of co-location, elevating the impact of optimal co-location – the focus of this work.

A. Co-Location

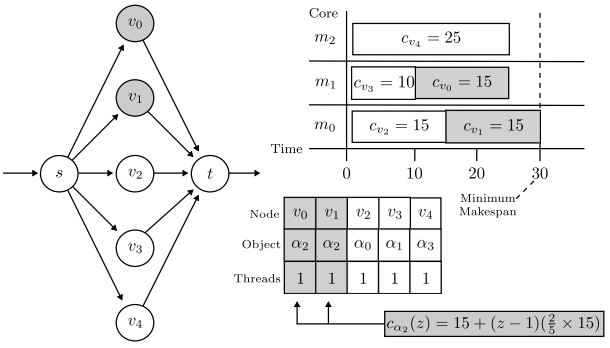
Within the Fork-Join model there is no distinction between a thread and an object; they are combined in the concept of a node. A node represents a thread executing an object, a thread may execute upon any of the available cores.

To *co-locate* multiple threads, the distinct requests that may span multiple cores are combined into a single request that must execute upon a single core according to the BUNDLE scheduling algorithm. Only threads that share an object may be co-located. When co-located, the combined WCET of all threads is no greater than the sum of their independent contributions (Property 1).

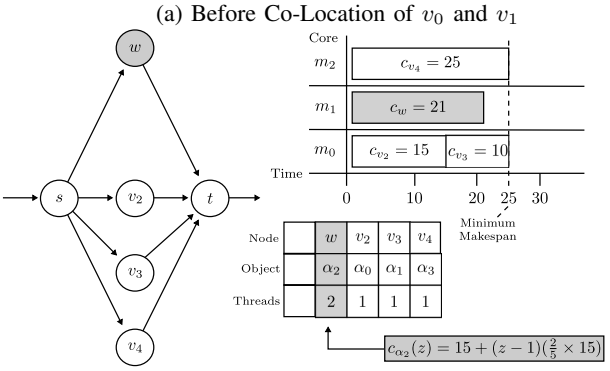
To support co-location within Fork-Join tasks, the graph within the model is augmented. A task's graph is a tuple $G = (V, E, O)$, containing a set of nodes V, edges $E \in V \times V$, and a set of objects O. In place of a single WCET C_v, every node $v \in V$ is given two attributes: an executable object $v.\alpha \in O$ and a number of threads $v.z \in \mathbb{N}$. Each node represents the uninterrupted execution of $v.z$ threads of the object $v.\alpha$ upon a single core according to the BUNDLE scheduling algorithm. The WCET of $z \in \mathbb{N}$ threads of the object $\alpha \in O$ scheduled by BUNDLE is given by $c_\alpha(z)$. As a convenience, the WCET of a node v is denoted by c_v and is equal to $c_{v.\alpha}(v.z)$. For any fork, fork-join, or join node $s \in V$, the number of threads is exactly one i.e. $s.z = 1$.

IV. Optimal Co-Location of Fork-Join Tasks

For Fork-Join tasks as described in Section II, individual nodes represent the co-located execution of multiple threads by the BUNDLE scheduling algorithm. For a specific task, additional threads may be co-located by joining the execution requests of distinct nodes contained within a parallel section into a single node.



(a) Before Co-Location of v_0 and v_1



(b) After Co-Location of v_0 and v_1

Fig. 2: Co-Locating v_0 and v_1 as w

To co-locate two nodes $u, v \in V$ of a Fork-Join task $G = (V, E, O)$, they must represent requests to execute the same object $u.\alpha = v.\alpha$. Joining the nodes u, v removes them

from the graph, inserting a node w: $V = (V \setminus \{u, v\}) \cup \{w\}$. The inserted node shares the executable object and total threads of the removed nodes: $w.\alpha = u.\alpha = v.\alpha$ and $w.z = u.z + v.z$. All incident edges of u and v are transitioned to w. By Property 1, $c_w \leq (c_u + c_v)$.

Reducing the total demand of a parallel segment may decrease its makespan, thus reducing the makespan of its task. Figure 2 illustrates the co-location operation and the potential reduction of the minimum makespan of a parallel section. The graph in each subfigure represents a parallel section before and after co-location. In the upper right of each subfigure is a GANTT chart representing a minimum makespan of the parallel section for a system with three cores. In the lower right, a small table identifies the objects associated with each node and the WCET bounding function for α_2.

Before co-location v_0 and v_1 contribute a total of 30 units to the makespan of the parallel section. After their co-location as w the total contribution is 21 units. Before co-location, the minimum makespan of the section is 30 units. Co-location reduces the minimum makespan to 25 units.

Co-location affects the structure of a task. It reduces the number of threads executing in parallel within a parallel section, potentially increasing the makespan of a task. Consequently, co-location may create an infeasible task from a (potentially) feasible one. Any algorithm that attempts to leverage co-location to reduce makespans or the number of cores required to meet a task's deadline must decide which threads to co-locate, leading to the optimization problem focused upon in this work.

Definition 5 (Optimal Co-Location of a Fork-Join Task). Given a fork-join task with graph G, an optimal co-location of G is the graph $\hat{G}$ obtained by co-locating nodes of G that is schedulable ($\Lambda \leq D$) requiring the fewest number of cores m.

Note Definition 5 of optimal co-location is compatible with the definition of *optimal collapse* for DAG tasks found in [70]. Since all Fork-Join tasks are DAG tasks, the complexity of optimal collapse of a DAG task is no less than the complexity of optimal co-location of a Fork-Join task.

An iterative approach is proposed to calculate the optimal co-location of a Fork-Join task. Starting with one core $m = 1$, the minimum makespan Λ achievable through co-location is calculated. If the task is not schedulable ($\Lambda > D$), m is increased by 1 and the makespan recalculated. Iteration ceases when the task is schedulable or when m exceeds the cores available on the system $\mathbb{M} \in \mathbb{N}$. The smallest $m \leq \mathbb{M}$ for which $\Lambda \leq D$ reflects an optimal co-location of the task.

Pseudocode of the iterative algorithm is provided in Algorithm 1. Each iteration invokes MIN-TSPAN for a specific number of cores m to determine the minimum makespan for the task G. Leveraging Observation 2, MIN-TSPAN invokes MIN-PSPAN to independently calculate the minimum makespan of each parallel section achievable through co-location. Summing the minimum makespans of the parallel sections and WCET of all sequential nodes yields the makespan of the task Λ. The complexity of optimal co-location OPT-CORE is determined by

Algorithm 1 OPTCORE

```
1: procedure OPT-CORE(G, D, M)
2:     m ← 1
3:     while m ≤ M do
4:         d ← MIN-TSPAN(G, m)
5:         if d ≤ D return m
6:         m ← m + 1
7:     end while
8:     return fail
9: end procedure
10: procedure MIN-TSPAN(G, m)
11:     Derive S and P from G
12:     d ← Σ_{s∈S} c_s
13:     for p ∈ P do
14:         d ← d + MIN-PSPAN(p, m)
15:     end for
16:     return d
17: end procedure
```

the complexity of MIN-PSPAN due to the polynomial $\mathbb{O}(\mathbb{M}|P|)$ invocations of MIN-PSPAN. By virtue of the structure of OPT-CORE and Observation 2 the focus of the co-location problem and its complexity is placed upon minimizing the makespan of parallel segments (MIN-PSPAN).

V. Minimum Makespan Schedules of Individual Parallel Sections

By Definition 1, the makespan λ of a parallel section P is an upper bound on the time required to execute all threads of all nodes given m cores. Nodes (and their co-located threads) may be executed in any order, independent of other nodes within the parallel section. Intuitively, this creates two interdependent problems when calculating the minimum makespan of a parallel section. The first problem is the selection of which nodes to co-locate. The second is the assignment and execution order of nodes upon the m cores.

In this section, the minimization problem will be simplified from two problems to one by introducing *minimum upper bound schedules*. The problem of co-located scheduling upon m cores is phrased as the familiar makespan problem.

Parallel nodes within a parallel section have a single dependency, the preceding fork node. Each node may be executed in any order, independent of other nodes within the parallel section. The assignment, order, and co-location of all threads to cores is referred to as a *parallel section schedule*. For simplicity, the set of objects within a parallel section is denoted A. A parallel section schedule is comprised of m individual *core schedules* denoted $\widetilde{H}_1, \widetilde{H}_2, ..., \widetilde{H}_m$. A core schedule is an ordered list of nodes, represented by the node's object and thread count, the elements are denoted $\alpha \cdot z$ where $\alpha \in A$ is the object and $z \in \mathbb{N}$ is the number of co-located threads of α. A core schedule represents the ordered execution of sets of co-located threads, e.g. $\widetilde{H}_2 = (\alpha_2 \cdot 2, \alpha_1 \cdot 1, \alpha_3 \cdot 5)$. The length of a core schedule L_i is the sum of its co-located execution times. When a core schedule $\widetilde{H}_i$ or length L_i includes a subscript, the subscript identifies a core ie. $1 \leq i \leq m$.

Definition 6 (Length of a Schedule L). For a schedule $\widetilde{H}_i = (\alpha_1 \cdot z_1, \alpha_2 \cdot z_2, ...)$, the length of the schedule L_i is the sum

of the co-located threads WCET values:

$$L_i \leq \sum_{\alpha \cdot z \in \widetilde{H}_i} c_\alpha(z) \tag{4}$$

Execution of the join node following a parallel section P cannot begin until all parallel nodes of P complete their execution. Nodes outside of P cannot participate in any of the m core schedules. Therefore, the longest core schedule bounds the makespan λ of the parallel section P.

Definition 7 (Makespan λ of P Given m Cores)*.* For a parallel section P, a system with m cores, the makespan λ of P is the maximum length of any of the core schedules of P:

$$\lambda = \max_{i \in \{1,2,\ldots,m\}} L_i \tag{5}$$

For a given core schedule $\widetilde{H}$, if threads of α appear in multiple elements, co-locating all threads of α into a single element reduces the length of its schedule L.

Theorem 1 (Minimum Upper Bound of a Schedule)**.** Given a core schedule $\widetilde{H} = (\alpha_1 \cdot z_1, \alpha_2 \cdot z_2, ...)$ the minimum upper bound length of $\widetilde{H}$ is the length of the schedule H which co-locates all threads of identical objects in $\widetilde{H}$.

Direct Proof: Consider a schedule $\widetilde{H}$ with distinct elements for an object α, assign the first occurrence of α index i and the second occurrence index j. The two elements of α are labeled $\alpha_i \cdot z_i$ and $\alpha_j \cdot z_j$.

Let $k = |\widetilde{H}| + 1$, $\alpha_k = \alpha$, and H be the core schedule of $\widetilde{H}$ after co-locating the two elements of α:

$$H = \left(\widetilde{H} \setminus \{\alpha_i \cdot z_i, \alpha_j \cdot z_j\}\right) \cup \{\alpha_k \cdot (z_i + z_j)_k\}$$

By Property 1 (Joined Request Bound):

$$c_\alpha(z_i + z_j) \leq c_\alpha(z_i) + c_\alpha(z_j)$$
$$\therefore\ L_H \leq L_{\widetilde{H}}$$

Since α was selected arbitrarily, co-locating all threads of identical objects of $\widetilde{H}$ in H minimizes L_H. ∎

Corollary 1 (Minimum Combined Execution Time)*.* Given a set of nodes represented by their object and thread counts $\widetilde{V} = \{\alpha_1 \cdot z_1, \alpha_2 \cdot z_2, ...\}$, the minimum total WCET of all threads W is the total WCET of V, where all threads of identical objects of $\widetilde{V}$ are co-located in V:

$$W = \sum_{\alpha \cdot z \in V} c_\alpha(z)$$

Proof by Substitution: Within Theorem 1, substitute W for L, $\widetilde{V}$ for $\widetilde{H}$, and V for H. ∎

The conversion from a core to a minimum upper bound schedule is denoted by a function $\texttt{MUB}(\widetilde{H})$. Figure 3 provides an example. Conversion, in a straightforward manner, is an $\mathbb{O}(|\widetilde{H}|^2)$ operation. Note, the object order in a minimum upper bound schedule does not impact the length of the schedule.

$H \leftarrow \texttt{MUB}(\widetilde{H})$	$\widetilde{H}$
$\{\alpha_1 \cdot 2, \alpha_3 \cdot 1, \alpha_2 \cdot 5\}$	$\{\alpha_1 \cdot 1, \alpha_3 \cdot 1, \alpha_2 \cdot 4, \alpha_1 \cdot 1, \alpha_2 \cdot 1\}$

Fig. 3: Core Schedule to Minimum Upper Bound Schedule

The complexity of calculating the minimum makespan of a parallel section is determined by a pair of problems: PARALLEL SECTION SCHEDULING (PARS) and PARALLEL SECTION MINIMUM MAKESPAN (PARM). The PARS problem decides for a parallel section P whether or not all threads can complete upon m cores before a deadline D_P. The Fork-Join model does not require, nor include a deadline D_P per parallel section. The deadline is included to form a decision problem. For descriptive ease the set of objects A are ordered $A = \{\alpha_1, \alpha_2, \ldots, \alpha_{|A|}\}$. The ordering is shared with the set of threads $Z = \{z_1, z_2, \ldots, z_{|Z|}\}$, where $z_i \in \mathbb{N}$ is the total number of threads of $\alpha_i \in A$ in P. The total threads of a parallel section are given by $\xi = \sum_{z_i \in Z} z_i$.

Problem 1 (PARALLEL SECTION SCHEDULING (PARS))**.** Given a system with $m \in \mathbb{N}$ cores, a parallel section P, deadline D_P, set of ordered objects A, total set of ordered threads Z, is there a partition of threads into m distinct subsets represented as minimum upper bound schedules $H_1, H_2, ..., H_m$ such that the makespan λ of P is no greater D_P:

$$\lambda = \max_{1 \leq i \leq m} \left\{ \sum_{\alpha \cdot z \in H_i} c_\alpha(z) \right\} \leq D_P$$

Problem 2 (PARALLEL SECTION MINIMUM MAKESPAN (PARM))**.** Given an instance I of PARS, PARM is the least value of D_P for which PARS decides "yes" for I.

The PARM problem describes the operation of MIN-PSPAN within OPT-CORE (Algorithm 1). Given PARM is at least as hard as PARS, the more convenient problem of the two will be used to analyze the complexity of both. Being polynomially related to OPT-CORE, the complexity of PARM determines the complexity of OPT-CORE. Both PARS and PARM problems are shown to be NP-Hard in the strong sense by reducing the strongly NP-Hard problem of MULTIPROCESSOR SCHEDULING [21] to PARS.

Problem 3 (MULTIPROCESSOR SCHEDULING)**.** Given a set of tasks A, lengths $l(a) \in \mathbb{N}$ for all $a \in A$, and deadline $D \in \mathbb{N}$ is there a partition $A = A_1 \cup A_2 \cup ... \cup A_m$ of m disjoint sets such that Equation 6 holds?

$$\max_{1 \leq i \leq m} \left\{ \sum_{a \in A_i} l(a) \right\} \leq D \tag{6}$$

Intuitively, the reduction creates an instance of PARS by adding one thread of each object for each length $l(a)$.

Theorem 2 (PARS is NP-Hard)**.** PARS is NP-Hard in the strong sense.

Proof by Reduction from MULTIPROCESSOR SCHEDULING*:* Given an instance of MULTIPROCESSOR SCHEDULING. For every task $a_i \in A = \{a_1, a_2, \ldots, a_{|A|}\}$ let

1) $\alpha_i = a_i$
2) $z_i = 1$
3) $c_{\alpha_i}(1) = l(a_i)$

Every task $a_i \in A$ from MULTIPROCESSOR SCHEDULING is mapped to a distinct object α_i in PARS. Therefore, no nodes may be co-located in PARS. The individual core schedules $H_1, ..., H_m$ are equal to their minimum upper bound schedules. Hence a partition of A from MULTIPROCESSOR

SCHEDULING is a set of minimum upper bound core schedules $H = H_1, ..., H_m$ in PARS i.e. $A = H$. Since A is unmodified, if A is a partition that satisfies MULTIPROCESSOR SCHEDULING with a makespan less than or equal to D so will the core schedules of H have a makespan less than or equal to D. ■

VI. Approximation

Being NP-Hard in the strong sense, a tractable exact algorithm for the PARM problem does not exist unless **P = NP**. In place of an exact algorithm, a 3-factor approximation algorithm (3-PARM) is proposed.

The approximation provides a 3-factor guarantee via the lower bound LB of a makespan for a parallel section P executing on $m \in \mathbb{N}$ cores. There are two possible values for LB. Co-locating all threads of each object produces the minimum combined WCET for a parallel section by Corollary 1. Averaging the minimum total WCET over the m available cores, every core schedule is of equal minimum length; serving as the first value for LB. The average may be smaller than one heavy weight thread of an object in A. Thus, the second possible value for LB is the heaviest heavy-weight thread. There must exist a *heaviest object* $h \in A$ such that $\forall_{\alpha \in A}\ c_h(1) \geq c_\alpha(1)$. No schedule may be shorter than $c_h(1)$. Combining the two possible values, the lower bound of PARM is given by the following equation.

$$LB = \max\left\{c_h(1), \frac{1}{m}\sum_{\alpha_i \in A} c_{\alpha_i}(z_i)\right\}$$

Presented as pseudocode in Algorithm 2, the process of calculating the approximate makespan is to iterate over the cores, assigning threads to one core until the length of the core's schedule exceeds the lower bound. Once exceeded, the next core is selected, the core schedule is filled with threads until exceeding the lower bound, and the process is repeated until all cores have been processed.

Threads are assigned according to their object. Every thread of one object is assigned before advancing to the next object. While iterating over each core $i \leq m$, the algorithm selects an arbitrary object $\alpha_j \in A$, adding one of its threads to the core's schedule $\widetilde{H}$ on Line 8. The *estimated* length of the schedule L_i is increased by the WCET of a heavy weight thread of α_j on Line 9. For the remaining $z_j - 1$ threads, each is added in turn to $\widetilde{H}$ as a distinct execution request and to L_i as a light weight thread until the length exceeds the lower bound: $L_i > LB$. Upon exceeding LB if all threads of α_j have not been assigned, i is incremented, where the remaining threads are added to $\widetilde{H}$ and light threads of α_j are added to the new L_i until LB is exceeded or all threads have been assigned.

Lines 8 and 12 of 3-PARM assign one thread of α_j to some schedule $\widetilde{H}_i$. Once assigned to a schedule all threads for α_j within $\widetilde{H}_i$ will be co-located according to Theorem 1 on Lines 18-21. The maximum of the minimum upper bound schedules H_{max} is returned as the makespan of the section.

Theorem 3 (3-PARM is in **P**)**.** 3-PARM is $\mathbb{O}(\xi)$.

Direct Proof: Threads are assigned to core schedules serially starting with the for loop on line 7. Thus, the complexity is bounded by the total number of threads in the parallel section ξ. Hence, 3-PARM is $\mathbb{O}(\xi)$ and in **P**. ■

Algorithm 2 3-PARM

```
1: procedure 3-PARM(P = (A, V), m)
2:     H̃ ← ∅ × m                                  ▷ m empty schedules
3:     L ← 0 × m                                  ▷ m estimated lengths of 0
4:     i ← 1
5:     while i ≤ m ∧ |A| > 0 do
6:         a_j ← an element of A
7:         A ← A \ a_j
8:         H̃_i ← H̃_i ∪ (α_j · 1)
9:         L_i ← β_{α_j}                          ▷ Heavy request
10:        k ← 1
11:        while k < z_j do                       ▷ Light requests
12:            H̃_i ← H̃_i ∪ (α_j · 1)
13:            L_i ← L_i + (γ_{α_j})(β_{α_j})
14:            k ← k + 1
15:            i ← i + 1 if L_i > LB
16:        end while
17:    end while
18:    for H̃_i ∈ H̃ do
19:        H_i ← MUB(H̃_i)                         ▷ Theorem 1
20:        H ← H ∪ H_i
21:    end for
22:    H_max = argmax_{H_i ∈ H} L_{H_i}
23:    return H_max
24: end procedure
```

Within 3-PARM, the estimated length L_i of each schedule H_i may be shorter than the actual length. When assigning the z_j threads of an object a_j, exactly one heavy weight thread is added to precisely one core schedule H_{i-1}. During the execution of the `while` loop beginning on line 11, if L_{i-1} exceeds LB before all z_j have been assigned, H_i will be assigned at least one light weight thread of a_j and no heavy weight threads. Therefore, L_i will be β_{α_j} units shorter than the actual minimum upper bound schedule. Theorem 4 utilizes the estimated lengths L_i to show the actual length of the longest schedule H_{max} provides the 3-factor guarantee.

Theorem 4 (3-PARM guarantee)**.** 3-PARM yields an approximation guarantee of 3.

Direct Proof:

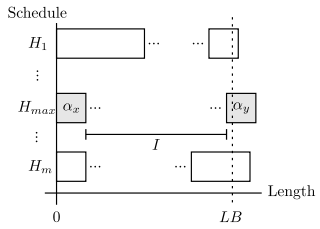


Fig. 4: H_{max} and L_{max}

Upon completion, 3-PARM returns the greatest minimum upper bound schedule H_{max} with length L_{max}. Denote the estimated length and core schedule of H_{max} as L_{est} and H_{est} respectively where $H_{max} = \texttt{MUB}(H_{est})$. Within H_{max} denote the first and final elements $\{\alpha_x \cdot 1\}$ and $\{\alpha_y \cdot 1\}$ respectively.

Illustrated by Figure 4, the contributors to length L_{max} may be decomposed into three components, $\{\alpha_x \cdot 1\}$, $\{\alpha_y \cdot 1\}$, and the intermediate threads I between α_x and α_y: $I = H_{est} \setminus \{\alpha_x \cdot 1, \alpha_y \cdot 1\}$.

By construction, the contribution of I to L_{est} and L_{max} are equal. The first element of I is the second thread assigned to H_{max}, denote this element $\alpha_i \cdot 1$. It is the case that $\alpha_i = \alpha_x$ or it does not. If $\alpha_i = \alpha_x$ then it will contribute a light

weight cost to H_{est} (Line 12) and H_{max} after conversion to a minimum upper bound schedule. If $\alpha_i \neq \alpha_x$, it will contribute a heavy weight cost to H_{est} (Line 9), and a heavy weight cost to H_{max} after conversion to a minimum upper bound schedule. The remaining elements of I contribute equally to L_{est} and L_{max} by the same reasoning. Further, by construction, the contribution of I must be less than LB, ie. $L_I \leq LB$.

By construction, α_x may have contributed a light weight value to L_{est}. However, within H_{max}, α_x may be a heavy weight thread of the heaviest object contributing a heavy weight to L_{max}: $c_{\alpha_x}(1) \leq c_h(1) \leq LB$. Similarly, α_y may be one thread of the heaviest object: $c_{\alpha_y}(1) \leq c_h(1) \leq LB$.

Thus, the length L_{max} is upper bounded by:

$$\begin{aligned} L_{max} &\leq c_{\alpha_x}(1) + L_I + c_{\alpha_y}(1) \\ &\leq LB + LB + LB \leq 3LB \end{aligned}$$

Since the optimal scheduler cannot produce a makespan less than the lower bound LB, and $L_{max} \leq 3LB \leq 3$PARM, 3-PARM is a 3-factor approximation. ∎

A. Resource Augmentation Bound

An approximation guarantee of parallel section makespans does not fully inform the schedulability of tasks comprised of parallel sections and sequential nodes. A scheduling algorithm with a resource augmentation bound $B \geq 1$ successfully schedules a task on m processors of speed B if an optimal scheduling algorithm can successfully schedule the task by its deadline on m processors of speed 1. For each parallel section, 3-PARM has a resource augmentation bound of $B = 3$, further the bound applies to the scheduling of the complete task. The two following theorems (Theorems 5 and 6) establish the bound for parallel sections and Fork-Join tasks scheduled utilizing the co-located schedules of 3-PARM.

Theorem 5 (Parallel Section Resource Augmentation Bound for 3-PARM)**.** 3-PARM has a resource augmentation bound of $B = 3$ for parallel sections.

Direct Proof: For a parallel section P, executing on m cores, the makespan of an optimal algorithm is termed OPT. The lower bound LB of 3-PARM is defined as the equal distribution of WCET of all threads across m cores when all threads are co-located. Since the total WCET is minimized by co-locating all threads, and the total is equally distributed, LB must also be a lowerbound on the makespan of an optimal schedule: $LB \leq$ OPT.

The upper bound of a makespan for a parallel section schedule generated by 3-PARM is $3LB$. Executing a parallel schedule generated by 3-PARM on a processor of speed $B = 3$ will complete in $\frac{3LB}{B} = LB \leq$ OPT, which is less than or equal to the optimal makespan. Therefore, 3-PARM has a resource augmentation bound $B = 3$ for parallel sections. ∎

Betwixt parallel sections of Fork-Join tasks lie sequential nodes. According to the Fork-Join model, execution of sequential nodes must be executed in isolation (sans parallelism). As such, there can be no significant difference between an optimal scheduling algorithm's execution of sequential nodes and an algorithm that schedules parallel sections by 3-PARM.

Theorem 6 (Fork-Join Task Resource Augmentation Bound for 3-PARM schedules of parallel sections)**.** Scheduling parallel sections of a Fork-Join task with 3-PARM has a resource augmentation bound of $B = 3$.

Direct Proof: By Definition 2 the makespan of a Fork-Join task is divided into the contribution of sequential nodes and parallel sections:

$$\Lambda = \underbrace{\left(\sum_{s \in S} c_s\right)}_{\text{sequential nodes}} + \underbrace{\left(\sum_{P_i \in P} \lambda_i\right)}_{\text{parallel sections}}$$

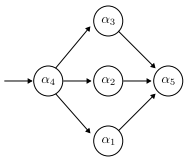
Denote the makespan of an optimal scheduler and the schedule utilizing 3-PARM as Λ_{OPT} and Λ_{3P}, respectively. For each parallel section $P_i \in P$, denote the makespans calculated by 3-PARM and an optimal scheduler as λ_i^{3P} and λ_i^{OPT}, respectively. Consider the execution of the optimal schedule on m processors of unit speed and 3-PARM on m processors of speed 3. By Theorem 5, every makespan $\frac{1}{3}\lambda_i^{3P} \leq \lambda_i^{\text{OPT}}$. Incorporating Definition 2:

$$\begin{aligned} \Lambda_{\text{OPT}} - \frac{\Lambda_{3P}}{3} &\geq \left(\sum_{s \in S} c_s\right) + \left(\sum_{P_i \in P} \lambda_i^{\text{OPT}}\right) \\ &\quad - \frac{1}{3}\left(\left(\sum_{s \in S} c_s\right) + \left(\sum_{P_i \in P} \lambda_i^{3P}\right)\right) \\ \Lambda_{\text{OPT}} - \frac{\Lambda_{3P}}{3} &\geq \sum_{s \in S} c_s - \frac{\sum_{s \in S} c_s}{3} \\ \Lambda_{\text{OPT}} - \frac{\Lambda_{3P}}{3} &\geq \frac{2 \cdot \sum_{s \in S} c_s}{3} \geq 0 \end{aligned}$$

Meaning, Λ_{3P} running on processors of speed 3 will complete in equal or less time than Λ_{OPT}. Therefore, scheduling parallel sections of a Fork-Join task with 3-PARM has a resource augmentation bound of $B = 3$. ∎

B. 3-PARM-HD

The 3-PARM algorithm performs poorly given *pathological* parallel sections. Pathological sections exceed the task's deadline regardless of the number of cores. Consider the following example task with deadline $D = 32$ and one parallel section, where all objects have a single thread of execution:



A	$c(1)$
α_1	8
α_2	5
α_3	20
α_4	6
α_5	4

Fig. 5: Pathological Task for 3-PARM D=32

Due to α_3, the lowerbound of the section is 20. Abiding by 3-PARM's assignment of threads to cores, if α_1 or α_2 are assigned before α_3 the length of the first core's schedule will always be less than the lowerbound before assigning α_3. Therefore, α_3 will be assigned to the first core's schedule, resulting in a section makespan of 25, 28, or 33, task WCET of 35, 38, or 43, and an unschedulable task. Increasing the number of cores (infinitely) will not reduce the WCET. Yet,

the task is schedulable with two cores by assigning α_3 to distinct cores from α_1 and α_2.

To counteract pathological tasks, 3-PARM is modified with a heuristic deadline d and named 3-PARM-HD. The heuristic deadline is a parameter to the core schedule assignment (CSA) that constricts the length of core schedules. For 3-PARM when assigning a thread to a core schedule H_i, if the estimated length L_i exceeds LB, the thread is assigned to the next core schedule H_{i+1}. For 3-PARM-HD, the next core schedule is selected when the actual length exceeds the heuristic deadline d. Due to the heuristic deadline, CSA may exhaust the supply of cores, indicating the deadline cannot be met with m cores – unlike 3-PARM which does not return failure.

3-PARM-HD takes an iterative approach to finding the smallest heuristic deadline d for which the section is schedulable on m cores. A binary search for d explores the range $d \in [LB, 3 \cdot LB]$. If core schedule assignment returns success for a given d, then the search continues by reducing d. If core schedule assignment returns failure for a given d, the search continues by increasing d. The smallest possible value of d is taken as the makespan of the parallel section. Since the largest possible value for d is LB, 3-PARM-HD retains the 3-factor approximation ratio and resource augmentation bound of 3 from 3-PARM.

Algorithm 3 3-PARM-HD

```
1: procedure 3-PARM-HD(P = (A, V), m)
2:     low ← LB, high ← 3 · LB
3:     while low ≠ high do
4:         d ← (low+high)/2
5:         H_max ← CSA(P, m, d)
6:         if H_max ≠ FAILURE then
7:             high ← d
8:         else
9:             low ← d
10:        end if
11:    end while
12:    return H_max
13: end procedure
14: procedure CSA(P, m, d)
15:    H̃ ← ∅ × m, L ← 0 × m, i ← 1
16:    while i ≤ m ∧ |A| > 0 do
17:        a_j ← an element of A, A ← A \ a_j
18:        k ← 1
19:        H̃_i ← H̃_i ∪ (α_j · 1), L_i ← β_{α_j}        ▷ Heavy request
20:        while k < z_j do                              ▷ Light requests
21:            H̃_i ← H̃_i ∪ (α_j · 1), L_i ← L_i + (γ_{α_j})(β_{α_j})
22:            k ← k + 1
23:            if L_i > d then
24:                i ← i + 1
25:                goto Line 19                          ▷ L_i = actual length of H_i
26:            end if
27:        end while
28:    end while
29:    return FAILURE if i > m                           ▷ Error, unable to meet d
30:    max ← argmax_{i∈|L|} L_i
31:    return MUB(H_max)
32: end procedure
```

VII. Simulated Evaluation

Simulated evaluation of 3-PARM and 3-PARM-HD are presented as comparisons (1) to the exact solutions with co-location ExactColo and without ExactNoColo (2) to Graham's [22] 2-factor MULTIPROCESSOR SCHEDULING (MSCHED) approximation and (3) to the DAG [70] core allocation algorithms without co-location DAG-m and co-location heuristics DAG-LP and DAG-GB. The ExactColo and ExactNoColo methods yield the shortest makespan of parallel segments by exploring all possible schedules with and without co-location. Graham's greedy algorithm is applied to parallel segments without co-location.

Synthetic task generation is a randomized process. Task generation assigns a random number of objects, parallel sections, and deadline within limited ranges. Each parallel section is assigned a random number of threads, every thread is associated with random object from the task's set of objects. Every object α is assigned a WCET function $c_\alpha(z) = \beta_\alpha + (z-1)(\gamma_\alpha \beta_\alpha)$; β_α is randomly selected from a limited range and γ_α is randomly selected from a restricted percentage of β_α. Table I summarizes the generation parameters.

Group	$\lvert P\rvert$	$\lvert O\rvert$	ξ	β
E	[2, 4]	[2, 8]	[6, 12]	[25, 50]
X	[4, 8]	[8, 16]	[64, 256]	[50, 100]

Group	γ	D	n	M	$\lvert\tau\rvert$
E	$[5\%, 45\%]$	450	100	5	50000
X	$[10\%, 90\%]$	1800	500	64	50000

TABLE I: Synthetic Task Group Generation Parameters

Tasks are characterized by a metric termed the *cache reuse factor*. The cache reuse factor of a task quantifies, as a proportion, the maximum reduction in execution time achievable by co-locating all threads. The value ranges from $[0, 1)$, where a larger value reflects a greater potential to reduce WCETs and therefore makespans. The cache reuse factor of a task $\bar{F}(G)$ is defined by Equation 7, the average group factor $\bar{\mathbb{F}}$ is defined by Equation 8. For both equations, the set of parallel sections $P = \{P_1, P_2, ...\}$ and sequential nodes $S = \{s_1, s_2, ...\}$ are implicitly derived from G, the set of objects A and threads Z are implicitly derived from the in-scope parallel section.

$$\bar{F}(G) = 1 - \frac{\left(\sum_{s \in S} c_s\right) + \sum_{P_i \in P} \sum_{A \in P_i} \sum_{\alpha_j \in A} c_{\alpha_j}(z_i)}{\left(\sum_{s \in S} c_s\right) + \sum_{P_i \in P} \sum_{A \in P_i} \sum_{\alpha_j \in A} c_{\alpha_j}(1) \cdot z_j} \quad (7)$$

$$\bar{\mathbb{F}} = \frac{1}{|\tau|} \sum_{G \in \tau} \bar{F}(G) \quad (8)$$

Tasks are randomly generated according to the parameters of their group. Within a group tasks are filtered, removing trivially feasible and infeasible tasks. Once filtered, additional tasks are removed creating a distribution of tasks according to their $\overline{F}$ interval – the goal is to distribute an equal number of tasks per interval. As a subtractive process, reaching the target number of tasks per interval is not guaranteed. Details of construction, subtraction, and implementation, are available [66].

Two task groups are evaluated. The first E, is smaller and less complex, demonstrating the potential schedulability improvements of co-location. The second X, is larger and more complex, accentuating the benefit of co-location of the 3-PARM and 3-PARM-HD algorithms when compared to Graham's 2-factor MSCHED approximation and DAG heuristics.

Task group generation parameters are: $|P|$ the range of parallel sections per task, $|O|$ the number of objects per task,

ξ the number of threads per parallel section, β the range of base costs, γ the range of incremental costs, D the maximum deadlines, n the target number of tasks per $\bar{F}$ interval, M the maximium number of cores a task may utilize, and $|\tau|$ the number of tasks generated before filtering. With the exception of the incremental cost γ, the generation parameters were selected with the greatest range possible for each parameter with the intention of avoiding bias – while completing within 48 hours on the available hardware. The incremental cost γ ranges are informed by [68], where the incremental cost fell below 10% for benchmarks from the Mälardalen [27] suite, and are supported by experimentation in Section VIII.

The metrics of comparison for the algorithms are: the number of tasks schedulable on m cores, average completion time, and the number of cores required for a task to be schedulable. Table I lists the generation parameters for the "exact" group E and the "approximate" group X.

Figure 6 summarizes the results of the exact and approximate algorithms when applied to the smaller group of tasks E on 5 or fewer cores. The purpose of the graphs are to illustrate the trends of the approximations as they relate to the exact algorithms. Y-values represent the percentage of schedulable tasks within the cache reuse interval for each algorithm. Within this graph, the potential benefit of co-location for Fork-Join tasks is illustrated by the gap between the EXACTCOLO and EXACTNOCOLO algorithms. Comparing the PARM approximations to Graham's MSCHED approximation, 3-PARM performs poorly scheduling fewer tasks than Graham's. However, by incorporating a heuristic deadline 3-PARM-HD is able to outperform Graham's, scheduling 40% more tasks – implying pathological tasks are a significant impediment to 3-PARM. Further, 3-PARM-HD is able to schedule 95% of feasible tasks.

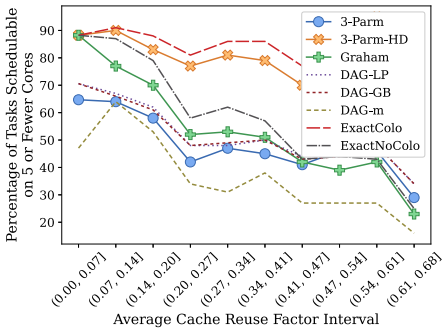


Fig. 6: Schedulability Results

By construction, trivially infeasible tasks (where the minimum WCET exceeds the deadline), and trivially feasible (where the total demand without co-location is less than the deadline) are removed. The consequence of subtracting such tasks is that tasks with low cache reuse factors are favorable with respect to schedualability for all algorithms. For E, the average cache reuse factor $\overline{\mathbb{F}}$ is .36.

When executing on an Intel Xeon Silver 4210R over E, the maximum running time of the EXACTCOLO and EXACTNOCOLO algorithms were 120 and 131 seconds respectively, while the maximum for the 3-PARM and 3-PARM-HD did not exceeded 0.02 seconds.

Figure 7 compares the core allocation performance of the algorithms. Only tasks deemed schedulable by all algorithms take part in the comparison.

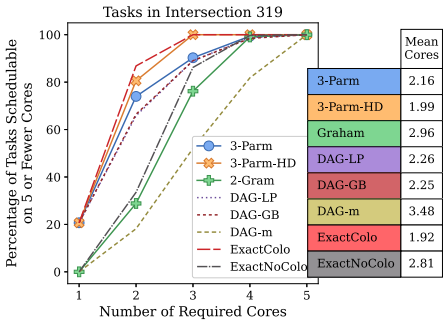


Fig. 7: Core Comparison for E

Of the 917 tasks of E, 319 were schedulable by all algorithms. 3-PARM and 3-PARM-HD allocated 27% and 32% fewer cores on average compared to Graham. On average the DAG heuristics require 4% or more cores than 3-PARM and 11% or more than 3-PARM-HD.

Shifting focus to the larger set X where construction produced an $\overline{\mathbb{F}}$ of 50.3%, the approximation algorithms were offered 64 cores to schedule tasks upon. Of the 3121 tasks, DAG-GB scheduled 7%, DAG-LP 8%, Graham 15%, 3-PARM 16% and 3-PARM-HD 44%.

The DAG heuristics perform poorly compared to the approximations, as they are unable to leverage the structure of Fork-Join tasks. The DAG heuristics prioritize nodes for collapse at the task level. DAG-GB co-locates nodes in decreasing order of the total workload decrease. DAG-LP co-locates node in decreasing order of critical path extension. They are unable to rely on the dedication of a core to a subset of co-located nodes as the approximation methods do.

For X, 423 of the tasks are schedulable by 3-PARM, 3-PARM-HD, and Graham. The DAG heuristics were omitted due to their poor performance.

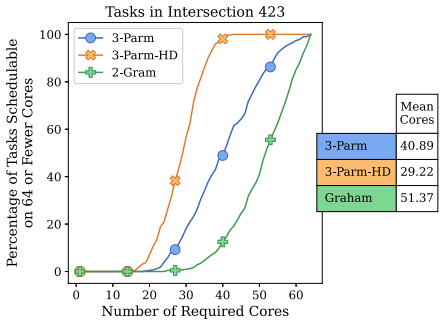


Fig. 8: Core Comparison for X

Figure 8 conveys the benefit of co-location upon core allocation, where 3-PARM requires 21% fewer and 3-PARM-HD requires 44% fewer cores than Graham.

Summarily, Figure 6 conveys the potential schedulability benefits of co-location as the gap between EXACTNOCOLO and EXACTCOLO. Through co-location, 3-PARM-HD outperforms Graham's approximation, scheduling nearly three times the number of tasks of X. Figure 8 shows among tasks deemed schedulable by both algorithms 3-PARM-HD requires 56% fewer cores than Graham's approach.

VIII. RISC-V EXPERIMENT

In addition to the simulation an empirical experiment was performed with a bare-metal Fork-Join scheduling algorithm implemented upon an 8-core RISC-V processor. The primary purpose of the experiment is to demonstrate the benefits of

the approximation algorithms and co-location on a realistic platform. The secondary purpose is to validate the parameters used in the synthetic evaluation. Source and installation instructions for the experiments are publicly available [66].

QEMU [23] provides the execution platform as an emulated 8-core RISC-V target. QEMU does not guarantee parallel execution of cores, nor do execution times reflect the impact of cache memory of the target system. As such, exact response times and makespans cannot be measured or calculated. Instead, per-core cycle counts are calculated through the number of instruction accesses and cache misses reported by the QEMU TCG cache plugin. To estimate makespans, the experiments limit tasks to a single parallel section, where the core with the greatest cycle count *represents* the task's makespan. Multiple parallel sections would obscure the representative value. Focusing upon a single parallel section does not detract from the validity of these results, as the approximation algorithms are also focused upon a single parallel section.

To calculate cycle counts, a memory model is required. The memory model of a modern RISC-V processor, the SiFive FE302 G002 [30] is used; a single 16kB 2-way instruction cache with 32B blocks and a random replacement policy. Memory accesses consume 2048 core clock cycles (e.g. $brt = 2048$). Instructions executions are modeled as a single cycle (e.g. $cpi = 1$).

When a Fork-Join executable terminates, the QEMU TCG cache plugin reports per core m the number of instructions accessed a_m and the number of cache misses b_m. The number of cycles a core executes c_m is the sum of access and reloading cycles due to misses: $c_m = a_m \times cpi + b_m \times brt$.

Benchmark	β	σ_β	γ	σ_γ
bs	88133	7.14	0.02	8.60e-05
bssort100	345426	7.62	0.78	4.20e-05
crc	201068	6.82	0.51	6.00e-05
expinit	79914	4.22	0.04	8.10e-05
fft	182636	6.54	0.07	4.80e-05
insertsort	74651	3.92	0.03	8.10e-05
jfdctint	172024	9.02	0.03	4.20e-05
lcdnum	58272	7.16	0.00	1.56e-04
matmult	485706	6.18	0.83	2.20e-05
minver	49836	6.24	0.00	1.86e-04
ns	86519	8.53	0.26	1.30e-04
nsichneu	1891047	6.35	0.70	6.00e-06
qurt	114525	10.03	0.01	9.10e-05
select	123072	9.15	0.03	7.90e-05
simple	53945	7.68	0.00	1.86e-04
sqrt	72769	5.93	0.01	1.04e-04
statemate	266232	6.45	0.01	2.80e-05
ud	137377	5.35	0.04	5.80e-05

TABLE II: MRTC Mean Base β and Incremental Costs γ

Previous BUNDLE [68] analysis utilized 18 of the Mälardalen WCET (MRTC) Benchmarks [27] . These 18 benchmarks were tested using the experimental platform to calculate representative base and incremental costs. To test a benchmark α, it is executed exactly once in isolation on one core m_i, then executed twice serially in isolation on a distinct core m_j, and the target is terminated. The cycles of m_i provide a representative value for the base cost of the benchmark α: $\beta_\alpha = c_{m_i}$. The difference in cycles of m_i and m_j is used to calculate the incremental cost: $\gamma_\alpha = \frac{c_{m_j} - c_{m_i}}{\beta_\alpha}$. These values are representative rather than exact due to the inclusion of non-deterministic synchronization costs. Table II summarizes the results of testing the 18 MRTC benchmarks 100 times, the β_μ and γ_μ values are means, with their standard deviations denoted σ_β and σ_γ.

Examining Table II, the representative incremental costs range from below 1 percent to 83 percent. Eleven of the benchmarks showed an incremental cost less than 5%. Only five of the benchmarks showed an incremental cost above 25%.

Combining the experimental data from Table II, the task set creation methodology from Section VII, and parameters from Table III, 1,000 tasks were generated. The 1,000 tasks are then filtered, removing tasks that are trivially feasible or infeasible, leaving 658 tasks in R. No further reduction of R is made.

Group	$\lvert P \rvert$	ξ	D
R	1	[64, 128]	3,200,000

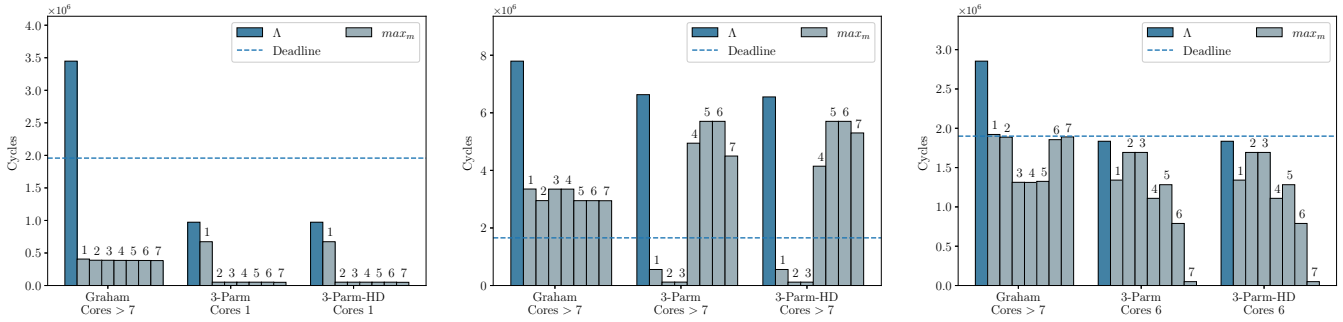
TABLE III: MRTC Group Generation Parameters

Analyzing R, the task cache reuse factor ranges from 18% to 95.3% with a mean $\overline{\mathbb{F}}$ of 55.8% and a standard deviation of 18.8%. There are two consequences of the observed reuse factors. First, the potential of co-location is verified for Fork-Join tasks on a RISC-V platform with a memory model matching the modern in-production FE302 chip. Second, the greater (+5%) average reuse factor than the X group supports the base and incremental costs in Table I, showing them to be conservative. Further, experimental results are **lower bounds**, calculated from the serial execution of objects rather than by a BUNDLE thread level scheduler.

From R three tasks were selected for comparative analysis and execution on the platform: FJ-791 with maximum reuse factor of .95, FJ-956 with the minimum reuse factor .18, and FJ-484 with reuse factor .55 (the mean). One of the eight cores of the experimental platform is reserved for synchronization. For seven cores, the approximation algorithms return a schedule and makespan for each of the tasks. For an unschedulable task, the approximation algorithms return their shortest 7 core makespan. The EXACTCOLO and EXACTNOCOLO algorithms were deemed impractical, as they were unable to return a schedulability result within 72 hours for any of the tasks.

The approximation schedules are manually converted to Fork-Join executables, assigning benchmarks to cores in the order prescribed by the schedule. Details of conversion are included in [66]. The executables are run 100 times, calculating the number of cycles per core. Across all runs, the maximum number of cycles observed on core m is denoted max_m.

Figure 9 summarizes the analytical and experimental results. Within each sub-figure, the Y-axis indicates the number of cycles and the dashed line the deadline of the task. Three groups separate the three approximations. Within a group, the analytical makespan is denoted by the Λ bar, the number of cores required to be schedulable below the approximation method's name (> 7 if unschedulable), and seven per-core worst case observed cycle counts with the core number m denoted atop each max_m bar.



(a) FJ-791 $\overline{F} = .95$ (b) FJ-956 $\overline{F} = .18$ (c) FJ-484 $\overline{F} = .55$

Fig. 9: Task Analysis and Experimental Results of Group R

The experiments validate the benefits of co-location. For task FJ-791, 3-PARM and 3-PARM-HD schedule a task on **one** core that Graham was unable to on seven. Further, FJ-791 highlights the pessimism of Graham's approach where the task's makespan is four times less than the calculated deadline. Task FJ-956 is unschedulable by any of the approximations. Given the low cache reuse factor and high utilization the result is unsurprising. For task FJ-484, Graham's MSCHED approximation deemed the task unschedulable on 7 cores, while the PARM approximations scheduled the task on 6 cores. Viewing the total area of the max_m bars per approximation, FJ-484 illustrates the impact of Corollary 1, the PARM approximations produce a smaller area compared to the MSCHED approximation, representing a reduction in execution time.

The experiments demonstrate the practical benefits of co-location executing established benchmarks upon a RISC-V platform. In the experiments, 3-PARM and 3-PARM-HD were able to reduce makespan and increase the number of schedulable tasks compared to Graham's.

IX. Related Work

Exact schedulability analysis of parallel scheduling algorithms is often intractable [50]. As such, schedulability analysis for Fork-Join parallel tasks continues to be an active area of research [1], [4], [15], [31], [37], [46], [51], [60], [74], [75]. However, these works are cache-agnostic, and do not account for the inter-thread cache benefit. At the time of writing, the authors are unaware of any preexisting Fork-Join schedulability analysis that incorporate ITCBs.

In the uniprocessor setting, accounting for the impact of cache memory upon real-time tasks is incorporated into WCET calculation [3], [40], [47], [48]. Accounting for cache memory may positively impact the non-preemptive scheduling of tasks by reducing WCET values. However, when tasks are scheduled preemptively response times may be negatively impacted due to cache-related preemption delay (CRPD) [2], [34], [38], [42]–[45], [63], [65], [71].

Alternative scheduling techniques have been developed to limit the negative impact of CRPD. The PREM [5], [54], [77] model and scheduling algorithm divides tasks into load and execute phases, limiting inter-task cache interference. Explicit preemption placement [9], [11], [38], [62], [76] permits preemptions at specific program locations to reduce their CRPD impact. These scheduling techniques take a negative perspective of caches, where inter-task (or inter-thread) cache interactions exclusively increase response times.

In the multi-processor setting for parallel tasks, shared caches increase the complexity of schedulability analysis and increase task response times – similar to CRPD. Bounding and mitigating evictions under global scheduling policies were examined in [12], [13]. Cache coherency and false sharing also extend completion times [18], [41], [58].

Caches are almost exclusively treated as a detractor to schedulability analysis by extending execution times. Few works treat cache memory positively. For uniprocessor single-threaded systems, *persistent cache blocks* share values between task releases [55], [56]. In the multi-processor setting, Calandrino [14] utilizes the *cache spread* in an empirical analysis.

A positive perspective of caches for multi-processor tasks was introduced in [70]. It merges federated multi-processor scheduling and BUNDLE [68] uniprocessor cache-aware scheduling algorithms through collapsing nodes within a DAG. In this work collapse is generalized to co-location. In [70], heuristics determined which nodes to co-locate; a complexity analysis of the optimal co-location problem was also absent.

Within this work, the optimal co-location problem closely resembles the multi-processor minimum MAKESPAN problem [21]. However, the optimal co-location problem differs significantly from MAKESPAN in that the order of tasks scheduled on a processor impacts the execution time of the task. Summarily, unlike MAKESPAN, the execution times of tasks in the optimal co-location problem are not independent.

X. Conclusion

In the previous sections, the complexity of optimal co-location is established as strongly NP-Hard for DAG and Fork-Join tasks. Due to the intractability of the problem, two approximations are presented as the first with guarantees and a resource augmentation bound. The approaches are able to schedule a greater number of tasks compared to Graham's MSCHED 2-factor approximation algorithm due to co-location. The practical benefits of co-location and the approximation algorithms are verified upon a RISC-V platform.

Acknowledgments

This work is supported by the National Science Foundation under grants CNS-2211640, CNS-2211641, & CNS-2306745.

REFERENCES

[1] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Helper locks for fork-join parallel programming. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 245–256.

[2] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[3] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. *Real-Time Systems Symposium, 1994., Proceedings.*, pages 172–181, Dec 1994.

[4] Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 215–224.

[5] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, Aug 2012. doi:10.1109/RTCSA.2012.48.

[6] Sanjoy Baruah. Feasibility analysis for hpc-dag tasks. *Real-Time Systems*, 58(2):134–152, 2022.

[7] Sanjoy Baruah. An ilp representation of a dag scheduling problem. *Real-Time Systems*, 58(1):85–102, 2022.

[8] Sanjoy Baruah and Alberto Marchetti-Spaccamela. Feasibility analysis of conditional dag tasks. In *Proceedings of the EuroMicro Conference on Real-Time Systems (ECRTS 2021)*., 2021.

[9] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 217–227, July 2011. doi:10.1109/ECRTS.2011.28.

[10] Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient real-time scheduling of dag tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(5):1–25, 2018.

[11] R. Bril, S. Altmeyer, M. van den Heuvel, R. Davis, and M. Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. *Real-Time Systems*, 53(4):403–466, July 2017.

[12] J. M. Calandrino and J. H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *2008 Euromicro Conference on Real-Time Systems*, pages 299–308, July 2008. doi:10.1109/ECRTS.2008.10.

[13] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 194–204, July 2009. doi:10.1109/ECRTS.2009.13.

[14] John Michael Calandrino. *On the Design and Implementation of a Cache-aware Soft Real-time Scheduler for Multicore Platforms*. PhD thesis, The University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2009.

[15] Daniel Casini. A theoretical approach to determine the optimal size of a thread pool for real-time systems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 66–78. IEEE, 2022.

[16] Peng Chen, Weichen Liu, Xu Jiang, Qingqiang He, and Nan Guan. Timing-anomaly free dynamic scheduling of conditional dag tasks on multi-core systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–19, 2019.

[17] Youngeun Cho, Dongmin Shin, Jaeseung Park, and Chang-Gun Lee. Conditionally optimal parallelization of real-time dag tasks for global edf. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 188–200. IEEE, 2021.

[18] Richard Cole and Vijaya Ramachandran. Analysis of randomized work stealing with false sharing. *Computing Research Repository - CORR*, 03 2011. doi:10.1109/IPDPS.2013.86.

[19] Zheng Dong and Cong Liu. An efficient utilization-based test for scheduling hard real-time sporadic dag task systems on multiprocessors. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 181–193. IEEE, 2019.

[20] Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Felipe Gohring De Magalhaes, Dahman Assal, and Gabriela Nicolescu. Cache locking content selection algorithms for arinc-653 compliant rtos. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019. doi:10.1145/3358196.

[21] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[22] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966. doi:10.1002/j.1538-7305.1966.tb01709.x.

[23] QEMU Software Group. QEMU. URL: https://www.qemu.org.

[24] Fei Guan, Long Peng, and Jiaqing Qiao. A fluid scheduling algorithm for dag tasks with constrained or arbitrary deadlines. *IEEE Transactions on Computers*, 2021.

[25] Fei Guan, Jiaqing Qiao, and Yu Han. Dag-fluid: A real-time scheduling algorithm for dags. *IEEE Transactions on Computers*, 70(3):471–482, 2020.

[26] Zhishan Guo, Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, and Nan Guan. Energy-efficient real-time scheduling of dags on clustered multi-core platforms. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 156–168. IEEE, 2019.

[27] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *International Workshop on Worst-Case Execution Time Analysis*, volume 15, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[28] Qingqiang He, Nan Guan, Zhishan Guo, et al. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

[29] Qingqiang He, Mingsong Lv, and Nan Guan. Response time bounds for dag tasks with arbitrary intra-task priority assignment. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[30] SiFive Inc. Documentation - SiFive, 2023. Freedom E310-G002 Manual, (Accessed May 5th, 2023). URL: https://www.sifive.com/documentation.

[31] Klaus Jansen, Oliver Sinnen, and Huijun Wang. An eptas for scheduling fork-join graphs with communication delay. *Theoretical Computer Science*, 861:66–79, 2021.

[32] Xu Jiang, Nan Guan, Haochun Liang, Yue Tang, Lei Qiao, and Wang Yi. Virtually-federated scheduling of parallel real-time tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 482–494. IEEE, 2021.

[33] Xu Jiang, Jinghao Sun, Yue Tang, and Nan Guan. Utilization-tensity bound for real-time dag tasks under global edf scheduling. *IEEE Transactions on Computers*, 69(1):39–50, 2019.

[34] Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1623–1628. EDA Consortium, 2007.

[35] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1), dec 2017. doi:10.1145/3092946.

[36] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *2010 31st IEEE Real-Time Systems Symposium*, pages 259–268, 2010. doi:10.1109/RTSS.2010.42.

[37] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, pages 259–268. IEEE Computer Society, 2010.

[38] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[39] Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

[40] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *17th IEEE Real-Time Systems Symposium*, pages 254–263, Dec 1996.

[41] Tongping Liu and Xu Liu. Cheetah: Detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 1–11, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854038.2854039.

[42] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 75–84, April 2013. doi:10.1109/RTAS.2013.6531081.

[43] Will Lunniss, Sebastian Altmeyer, Robert I Davis, et al. A comparison between fixed priority and edf scheduling accounting for cache related pre-emption delays. *LITES*, 1(1):01–1, 2014.

[44] Will Lunniss, Sebastian Altmeyer, Giuseppe Lipari, and Robert I Davis. Accounting for cache related pre-emption delays in hierarchical scheduling. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 183. ACM, 2014.

[45] Will Lunniss, Sebastian Altmeyer, Giuseppe Lipari, and Robert I Davis. Cache related pre-emption delays in hierarchical scheduling. *Real-Time Systems*, 52(2):201–238, 2016.

[46] C. Maia, P.M. Yomsi, and L Nogueira. Real-time semi-partitioned scheduling of fork-join tasks using work-stealing. *Journal of Embedded Systems*, 31, 2017. doi:https://doi.org/10.1186/s13639-017-0079-5.

[47] Frank Mueller. *Static Cache Simulation and Its Applications*. Ph.d. dissertation, Florida State University, 1995.

[48] Frank Mueller. Timing analysis for instruction caches. In *The Journal of Real-Time Systems 18*, pages 217–247, 2000.

[49] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, and Nathan Fisher. Tensity-aware optimized scheduling of parallel real-time tasks on multiprocessors. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8. IEEE, 2020.

[50] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10758, doi:10.4230/LIPIcs.ECRTS.2019.21.

[51] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama. Mouldable fork-join task scheduling techniques with inter and intra-task communications. *International Journal of Embedded Systems*, 15(1):69–81, 2022. doi:10.1504/IJES.2022.122074.

[52] OpenMP Architecture Review Board. OpenMP application program interface version 5.1, November 2020. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[53] Sims Hill Osborne, Joshua Bakita, Jingyuan Chen, Tyler Yandrofski, and James H Anderson. Minimizing dag utilization by exploiting smt. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 267–280. IEEE, 2022.

[54] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011. doi:10.1109/RTAS.2011.33.

[55] S. A. Rashid, G. Nelissen, S. Altmeyer, R. I. Davis, and E. Tovar. Integrated analysis of cache related preemption delays and cache persistence reload overheads. In *IEEE Real-Time Systems Symposium*, pages 188–198, Dec 2017. doi:10.1109/RTSS.2017.00025.

[56] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaut, and E. Tovar. Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In *Euromicro Conference on Real-Time Systems*, pages 262–272, July 2016. doi:10.1109/ECRTS.2016.25.

[57] Federico Reghenzani, Ashikahmed Bhuiyan, William Fornaciari, and Zhishan Guo. A multi-level dpm approach for real-time dag tasks in heterogeneous processors. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 14–26. IEEE, 2021.

[58] Suntorn Sae-eung. Analysis of false cache line sharing effects on multicore cpus. *Master's Projects*, 01 2010.

[59] Abusayeed Saifullah, Sezana Fahmida, Venkata P Modekurthy, Nathan Fisher, and Zhishan Guo. Cpu energy-aware parallel real-time scheduling. *Leibniz international proceedings in informatics*, 165, 2020.

[60] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. *ACM Transactions on Parallel Computing (TOPC)*, 6(4):1–33, 2019.

[61] Debabrata Senapati, Arnab Sarkar, and Chandan Karfa. Hmds: A makespan minimizing dag scheduler for heterogeneous distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–26, 2021.

[62] J. Simonson and J.H. Patel. Use of preferred preemption points in cache based real-time systems. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, 1995.

[63] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 2005 17th Euromicro Conference on Real-Time Systems (ECRTS)*, ECRTS '05, pages 41–48, July 2005. doi:10.1109/ECRTS.2005.26.

[64] Jinghao Sun, Nan Guan, Feng Li, Huimin Gao, Chang Shi, and Wang Yi. Real-time scheduling and analysis of openmp dag tasks supporting nested parallelism. *IEEE Transactions on Computers*, 69(9):1335–1348, 2020.

[65] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embededed Computing Systems*, 6(1), February 2007. doi:10.1145/1210268.1210275.

[66] Corey Tessler. Optimal co-location synthetic and experimental evaluation version 2.1. 2023. https://github.com/ctessler/fork-join-colo/releases/tag/v2.1.

[67] Corey Tessler and Nathan Fisher. Bundle: Real-time multi-threaded scheduling to reduce cache contention. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 279–290, 2016.

[68] Corey Tessler and Nathan Fisher. Bundlep: Prioritizing conflict free regions in multi-threaded programs to improve cache reuse. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 325–337, 2018.

[69] Corey Tessler and Nathan Fisher. NPM-BUNDLE: Non-Preemptive Multitask Scheduling for Jobs with BUNDLE-Based Thread-Level Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10752, doi:10.4230/LIPIcs.ECRTS.2019.15.

[70] Corey Tessler, Venkata Prashant Modekurthy, Nathan Fisher, and Abusayeed Saifullah. Bringing inter-thread cache benefits to federated scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[71] H. Tomiyama and N.D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*, pages 67–71, May 2000.

[72] Niklas Ueter, Mario Günzel, and Jian-Jia Chen. Response-time analysis and optimization for probabilistic conditional parallel dag tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 380–392. IEEE, 2021.

[73] Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. Reservation-based federated scheduling for parallel real-time tasks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 482–494. IEEE, 2018.

[74] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 83–94. ACM.

[75] Qi Wang and Gabriel Parmer. FJOS: practical, predictable, and efficient system support for fork/join parallelism. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 25–36. IEEE Computer Society, 2014.

[76] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real Time Computing Systems and Applications*, 1999.

[77] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013. doi: 10.1109/ECRTS.2013.26.

[78] Yaswanth Yadlapalli and Cong Liu. Lag-based analysis techniques for scheduling multiprocessor hard real-time sporadic dags. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 316–328. IEEE, 2021.

[79] Shuai Zhao, Xiaotian Dai, and Iain Bate. Dag scheduling and analysis on multi-core systems by modelling parallelism and dependency. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[80] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 128–140. IEEE, 2020.