Minimizing DAG Utilization by Exploiting SMT

Sims Hill Osborne, Joshua Bakita, Jingyuan Chen, Tyler Yandrofski, and James H. Anderson
Dept. of Computer Science, University of North Carolina, Chapel Hill, North Carolina, U.S.A.
{shosborn, jbakita, tylerdy, leochanj, anderson} @cs.unc.edu

***Abstract*—Parallel workloads are commonly modeled as directed acyclic graphs (DAGs). While DAG scheduling is an important tool, it is plagued by capacity loss; it is not uncommon to see half of a platform go unused. Here this loss is attacked from a new direction: reducing per-DAG utilization prior to assigning computing cores to a DAG. Specifically, simultaneous multithreading (SMT) is used to schedule individual nodes of a DAG task in parallel on the same physical computing core. An optimization program is given that applies SMT to a DAG in a way that minimizes total utilization without compromising correctness. Results for both individual DAGs and systems of DAGs are evaluated using both a large-scale study of synthetic DAGs and a case study. Optimal use of the program can reduce DAG utilization and required core counts by over 40% in the best cases and by 25% in nearly half of cases. Runtime requirements for the optimization program are considered, and a tunable parameter is provided to make tradeoffs between runtime and optimality, allowing even DAGs with 500 nodes to benefit.**

***Index Terms*—real-time systems, simultaneous multithreading, hard real-time, parallel scheduling algorithms, directed acyclic graphs, precedence constraints**

I. Introduction

Parallelism is an essential part of scheduling real-time workloads in modern applications such as image recognition [2, 18, 53], autonomous vehicles [30, 54], and aviation [37]. When the total processor time required for a task exceeds the task's relative deadline—a frequent occurrence—scheduling a task without parallelism becomes impossible.

Such tasks can be completed in a timely manner if they include segments that can be executed in parallel. These parallelizable workloads can be modeled using directed acyclic graphs, or DAGs. A DAG task is modeled in part as a graph $G = (V, E)$ with V denoting a set of vertices and E denoting a set of edges. Each vertex $v \in V$ represents a portion of the task, or *subtask*, and each directed edge (v_1, v_2) indicates that subtask v_1 must be completed before subtask v_2 can begin. Vertices that are not connected may be executed in parallel [6].

Unfortunately, scheduling DAGs tends to be inefficient. The well-known federated scheduling algorithm may see as much as half of a hardware platform go unused, and yet it is still a significant improvement over earlier methods for DAG scheduling [33]. There has been significant work in recent years aimed at reducing this capacity loss, some of which we discuss in Sec. II. Even so, the challenges associated with DAG scheduling remain a major obstacle to scheduling modern workloads in a safety-critical context.

We propose a new approach to parallel scheduling: scheduling individual subtasks in parallel on the same core using simultaneous multithreading (SMT). Using this method, it is possible to reduce both a DAG's utilization and core needs by nearly half without compromising safety.

Simultaneous multithreading. SMT is a technology that allows multiple programs to execute in parallel on a single computing core, reducing total execution time. SMT is capable of increasing the ability of a given hardware platform to schedule real-time systems [4, 12, 21, 22, 28, 39, 40, 41, 42, 45, 57].

In recent years, practitioners have used SMT in scheduling soft real-time [40, 42], hard real-time [39, 41], and mixed-criticality systems [4]. We advance the use of SMT in real-time systems by considering how to combine it with the DAG task model.

The following example shows the intuition behind using SMT to reduce a DAG's utilization.

Ex. 1. Consider the DAG shown in Fig. 1. It has a total execution cost of 130 time units; if the deadline is 110 time units, it has a utilization of $\frac{130}{110} \approx 1.18$. It has a *length*, or minimum execution time, of 70 time units; if it is allowed to execute on two processors, with v_3 and v_5 parallel to v_2 and v_4, it can complete in 70 time units.

Suppose that if subtasks v_2 and v_5 use SMT to execute on a single core, then 75 time units are required for both to complete. In that case, applying SMT to v_2 and v_5 produces the DAG shown in Fig. 2. All precedence constraints from the original DAG are maintained. The new DAG has a reduced total execution cost of 105 time units and a reduced utilization of $\frac{105}{110} \approx 0.95$. Furthermore, it can now be scheduled on a single physical core. ◀

Contribution and organization. We give an algorithm that applies SMT to a DAG task so as to minimize its utilization, as we did in Ex. 1. By doing so, we can make dramatic reductions in both the utilization and core requirements of DAGs. Our algorithm is optimal in terms of minimizing total utilization, but has exponential time complexity. To assist with using SMT with large DAGs, we include a tunable parameter that allows for tradeoffs between optimality and running time. We evaluate our

Supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

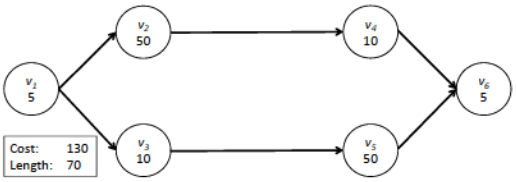


Fig. 1. A DAG task consisting of six subtasks.

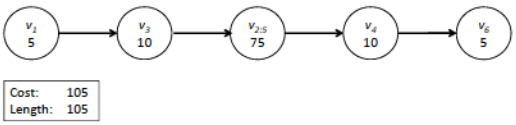


Fig. 2. The same DAG task; subtasks v_2 and v_5 paired via SMT.

work in terms of both individual DAGs and of systems of DAGs that use federated scheduling.

Our motivation is two-fold; first, particularly for DAGs that require many cores, minimizing utilization will tend to reduce the number of cores needed, making it possible to schedule DAGs that would otherwise be unfeasible on a given hardware platform. Second, reducing utilization can increase the amount of lower-priority work supportable in the mixed-criticality model of Vestal [50] or of other settings that allow multiple priorities of work to co-exist.

The remainder of this paper is organized as follows. In Sec. II, we cover background information, including an overview of SMT technology, DAG scheduling, a more precise explanation of some already-used terminology, and a full explanation of our system model.

In Sec. III we give our algorithm. This algorithm uses a quadratically constrained program (QCP) to decide which subtasks should be paired together to minimize a DAG's total utilization. We then show how the SMT-enabled DAG can be scheduled using existing methods, frequently using fewer cores than would otherwise be necessary.

In Sec. IV, we test our approaches via a large-scale synthetic task study that shows what reductions to utilization and core count can be expected in various scenarios; in some cases, we found that applying SMT can reduce a DAG's utilization by 40% and reduce the number of cores needed by nearly as much. In Sec. V we describe a case study that demonstrates that the schedules we produce are both implementable and safe. In Sec. VI, we conclude and suggest directions for future work.

II. Background

In this section, we give assumptions of our task and hardware platform models, provide an overview of SMT, explain how to schedule individual DAGs, and briefly cover existing methods for scheduling systems of DAGs.

A. Task and Platform Model

We consider the problem of scheduling a sporadic hard real-time task τ_i so as to minimize its utilization and, ideally, the number of cores[1] it requires. A task is defined as $\tau_i = (G_i, T_i)$, where $G_i = (V_i, E_i)$ is a DAG, as was discussed in Sec. I. T_i gives the period of τ_i, which releases a *DAG job* at most once every T_i time units, beginning at time 0. We assume *implicit deadlines*: every DAG job must complete within T_i time units of its release.

DAG-specific terminology. Each task τ_i consists of $|V_i|$ subtasks, with each subtask corresponding to a vertex in G_i. The *cost* of subtask v_k is given by c_k, assumed to be its worst-case execution time. The sum of all subtask costs gives the task's *total cost* C_i. We use capital letters to refer to characteristics of tasks and lowercase letters for characteristics of subtasks. When unambiguous, we omit subscripts from the capital letters. Each DAG job consists of one subjob for each subtask in the DAG. The DAG job is complete once all component subjobs have completed.

If within G there exists a path from vertex v_1 to v_2, then v_1 is a *predecessor* of v_2 and v_2 is a *successor* of v_1. Otherwise, they are *unconnected* and the corresponding subjobs within a single job can execute in any order, including simultaneously. If the vertices connect via a one-edge path, then the two are *immediate* predecessors and successors. We require that subtask indices follow a topological order, i.e. if $i < j$ holds, then v_j is not a predecessor of v_i.

Def. 1. A *chain* is a sequence of vertices in which all but the last are followed by one immediate successor. ◀

Def. 2. A DAG's total *length* L is equal to the maximum length of any chain in G. This value gives the minimum amount of time required to fully execute all jobs given an unlimited number of processors. ◀

Length is the formal term for the minimum task completion time time of 70 in Ex. 1. We require that $L \leq T$ holds for all tasks; otherwise, the task cannot be scheduled. While the term length is common in the DAG-scheduling literature, some sources use the term *critical path*.

A task's total *utilization* is defined as $U = \frac{C}{T}$. While purely sequential tasks must have $U \leq 1$ to be schedulable, DAG tasks with $U > 1$ are schedulable, given a sufficient number of cores, if $L \leq T$ holds [6]. We differentiate between *heavy* and *light* tasks.

Def. 3. τ_i is *heavy* if $U > 1$ holds and *light* otherwise. ◀

B. Overview of SMT Technology

In many modern processors, each core uses instruction-level parallelism within jobs to execute multiple instructions per cycle. By enabling SMT, this behavior is expanded to allow two or more jobs to execute instructions

[1]We use the word core exclusively to refer to *physical* cores, not the "logical cores" that may be provided by SMT.

within a single cycle. An overview of SMT execution is given in Ex. 2 and Fig. 3 below, closely following [41]. Details on the fundamentals of SMT can be found in the work of Eggers et al. [17], and a discussion of practical factors that can affect SMT execution in [9].

Ex. 2. Let v_1 and v_2 be unconnected subtasks of a DAG task. At the top of Fig. 3, jobs of v_1 (darker) and v_2 (lighter) execute sequentially without SMT on a core that can accept two instructions per cycle. When fewer than two instructions are ready, as in cycles 3 and 4, cycles are wasted. v_1 finishes at the end of 6 cycles and v_2 at the end of 12. In the second part of the figure, the same jobs employ SMT to execute in parallel, thereby reducing the number of lost cycles. v_1 finishes after 8 cycles and v_2 after 10. SMT thus delays the completion of v_1, but speeds up the completion of v_2 since it does not have to wait for v_1 to complete before beginning its own execution. ◀

Past research—our own and that of others— has shown that when SMT is in use, the majority of tasks execute at between 50% and 90% of their normal speed; in most cases, the execution time for two tasks in parallel with SMT is less than the time required to execute the two sequentially [4, 9, 10, 41, 42].

We limit our analysis to cores that can execute up to two tasks in parallel, as that is by far the most common design for SMT. It is currently implemented on Intel Xeon processors, under the name hyperthreading, and on AMD Zen processors; the latter is the platform for our case study.

SMT and safety. In previous work, we showed empirically that timing analysis for tasks using SMT can be as reliable as that for tasks without SMT [41]. Since then, we made further improvements by reducing the potential for SMT-specific cache interference [4]. We incorporate these earlier findings to make SMT reliable in our present work.

Specifically, we require that jobs using SMT must start simultaneously, as shown in Fig. 3; we have used this same rule in the past [4, 39, 41]. This restriction simplifies the timing analysis such that it is unnecessary to consider every possible combination of start times. We adapt this rule to our present work by limiting the application of SMT to *paired subtasks.*

Def. 4. If v_i and v_j form a *paired subtask*, denoted $v_{i:j}$, then the corresponding subjobs will always begin execution simultaneously on a single core. Paired subtasks must belong to the same DAG task, and each subtask may be in at most one pair. The paired subtask notation where $i = j$, i.e. $v_{i:i}$, is equivalent to v_i executing without SMT. ◀

Def. 5. [41] The *joint cost* for a paired subtask is given by the tuple $c_{i:j} = (c_{i(j)}, c_{j(i)})$, where $c_{i(j)}$ is the execution time of v_i given that it is paired with v_j and $c_{j(i)}$ is the reverse. If $i = j$, i.e. SMT is not used for v_i, then both elements of the tuple are equal to c_i. ◀

Ex. 3. To illustrate these joint costs, in Fig. 2, $c_{2(5)} = c_{5(2)} = 75$. In Fig. 3, $c_{1(2)} = 8$ and $c_{2(1)} = 10$. ◀

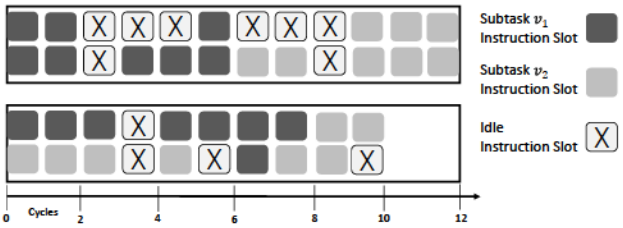


Fig. 3. Top: task execution without SMT. Bottom: execution with SMT.

We do not allow subtasks whose costs without SMT differ by a factor of 10 or more to be paired. Previous work [41] has found that in this case, timing becomes unreliable.

We note that many works on DAG scheduling make the implicit assumption that potential sources of interference in multicore systems can be safely dealt with. These interference sources include cache conflicts [1, 8, 14, 32, 36, 51, 52], DRAM conflicts [24, 26, 48, 55, 56], memory bus conflicts [38, 44], general OS support [3, 13, 27], and I/O conflicts [31, 43]. Potential cross-thread interference, which we explicitly address, is not fundamentally different from these other issues.

Revisiting cost and length. Given Def. 5, we formally define a task's total cost C and utilization U. This definition depends on how tasks are paired.

Def. 6. The total cost C and utilization U of a DAG are given by

$$C = \sum_{\text{for all pairs } (i,j)} \max(c_{i(j)}, c_{j(i)}) \text{ and } U = \frac{C}{T}.$$

Recall from Def. 4 that a subtask not using SMT is notationally considered to be paired with itself. ◀

Adding SMT to a DAG changes the definition of length only in that the vertices in a chain (Def. 2) may consist of either individual subtasks or of paired subtasks.

Ex. 4. In Fig. 3, v_1 completes at time 8. A successor of v_1 may begin execution at time 8 on another core (assuming it has no other incomplete predecessors). The successor task need not wait until v_2 finishes at time 10. ◀

C. Scheduling DAG Tasks.

Even without SMT, determining the minimum number of cores needed to schedule a heavy DAG is NP-hard in the strong sense [5] (light tasks can be scheduled sequentially on one core). To determine how many cores are needed by a heavy task, we follow Baruah [5] by using Alg. 1. In this algorithm, a job begins execution as soon as all of its predecessors have been allocated their WCETs and at least one core is free. We modify the algorithm by allowing "job" to mean a paired subjob; pairs may only begin once all predecessors of both component jobs are

complete. Alg. 1 has a speedup bound of $b = 2 - \frac{1}{m}$, i.e. if an optimal algorithm can schedule the DAG on m unit-speed cores, then Alg. 1 can schedule the same DAG on m speed b cores [5, 23].

Algorithm 1 Assign Cores [5, 23]

```
1:  Maintain ready as the set of all subjobs without incomplete predecessors.
2:  Maintain m_r as the number of cores not currently executing a subjob.
3:  m = ⌈U⌉
4:  Let t be the time and D the DAG deadline.
5:  while True do
6:     t = 0
7:     m_r ← m
8:     ready ← subjobs without predecessors
9:     while t ≤ D do
10:       while ready ≠ ∅ ∧ m_r > 0 do
11:          Assign ready subjobs to cores.
12:       end while
13:       t ← next subjob completion time
14:       if all subjobs are complete and t ≤ D then
15:          {m-core schedule found.}
16:          return m
17:       end if
18:    end while
19:    {No m-core schedule found; increment m.}
20:    m ← m + 1
21: end while
```

In research on scheduling systems of DAGs, federated scheduling on shared hardware is the most prominent approach. In this method, a system of DAG tasks is scheduled by executing each heavy task on a set of dedicated cores and partitioning light tasks among the remaining cores [33]. Heavy tasks may be over-assigned capacity; for example, a task needing two dedicated cores plus 10% of another core would be assigned three cores. For soft real-time tasks, where some deadline misses are allowable, cores can be reclaimed by work stealing, where idle cores "steal" work from busy ones [34].

Other methods that reduce the capacity loss of federated scheduling include semi-federated scheduling [29] and reservation-based federated scheduling [47]. In semi-federated scheduling, portions of heavy tasks are scheduled as if they were light tasks; the same task described above that would be assigned three cores under federated scheduling would be assigned two dedicated cores under semi-federated scheduling, with the remaining portion of the task scheduled as if it were independent. In reservation-based federated scheduling, DAG tasks are assigned reservation servers that can be scheduled sequentially.

III. Reducing DAG Utilization

In this section, we give our quadratically constrained program (QCP) for applying SMT to DAGs. The QCP minimizes total utilization[2] subject to the constraints that the DAG's total length is at most the DAG's deadline, paired subtasks begin simultaneously, and precedence constraints are respected.

QCP variables and constraints. We use the following variables to define our problem mathematically.

Def. 7. Let $x_{i:j}$ be defined to equal 1 if subtasks v_i and v_j are paired and 0 otherwise. For $i = j$, let $x_{i:j}$ equal 1 if v_i is to be executed without SMT and 0 otherwise. In all cases, we require that $x_{i:j} = x_{j:i}$ holds. We refer to these variables as the x variables. ◀

Def. 8. Let s_i and f_i give the start and finish times, respectively, for subtask v_i when the DAG, with a specified set of task pairs, is scheduled using Alg.1 with line 3 altered to make the initial value of m arbitrarily large. We refer to these variables, respectively, as the s and f variables. ◀

We can express a DAG's total utilization U in terms of x variables, joint costs, and periods. Eq. 1 restates Def. 6. Recall that V is the set of all vertices, each vertex corresponds to a subtask, and thus $|V|$ is the subtask count.

$$U = \sum_{i=1}^{|V|} \sum_{j=i}^{|V|} x_{i:j} \cdot \frac{\max(c_{i:j}, c_{j:i})}{T} \tag{1}$$

We define our QCP as minimizing Eq. 1 subject to the restrictions below.

(i) **Every subtask executes with at most one other subtask.** Recall that Def. 4 requires this restriction.

$$\forall i : \sum_{j=i}^{|V|} x_{i:j} = 1$$

(ii) **Paired subtasks begin simultaneously.** This restriction is required by Def. 4. For unpaired v_i and v_j, where $x_{i:j} = 0$, the expression below is trivially true.

$$\forall i, j : s_i \cdot x_{i:j} = s_j \cdot x_{i:j}$$

(iii) **All subtasks finish only after executing for sufficient time.** If v_i and v_j are paired, then v_i finishes at time $f_i = s_i + c_{i(j)}$. Notice that f_i is only dependent on s_i and $c_{i(j)}$, not on $c_{j(i)}$.

$$\forall i : f_i = s_i + x_{i:j} \cdot c_{i(j)}$$

(iv) **All subtasks finish prior to the deadline D.** This restriction preserves feasibility.

$$\forall i : f_i \leq D$$

[2]We have found that minimizing U is more practical than minimizing cores. Solving time is discussed in Sec. IV-B.

(v) **Precedence constraints are respected.** No subtask begins until all of its predecessors have completed.

$$\forall i : \forall j :: v_j \text{ precedes } v_i : s_i \geq f_j$$

Notice that (v) combined with (ii) prohibits precedence-constrained tasks from executing as pairs.

QCP output and effectiveness. The QCP outputs pairing decisions via the value of all x variables. Pairs can then be assigned to cores using Alg. 1. While the QCP is not optimal with respect to minimizing core count, we have found that for large-utilization DAGs, reductions in core count closely track reductions in utilization made using our methods. We further discuss this in Sec. IV.

Limiting the number of variables. The QCP's complexity is exponential on the number of variables, and the number of x variables needed is proportionate to $|V|^2$. Thus for large DAGs the QCP may be unmanageable. For those cases, we provide an additional restriction.

(vi) **Only subtasks with indices that differ by at most some constant K can be paired.**

$$|i - j| > K \rightarrow x_{i:j} = 0$$

In practice, not defining $x_{i:j}$ when $|i-j| > K$ provides a greater performance benefit than restricting $x_{i:j}$ to zero. Note that K can be at most $|V|$.

In our experiments, $K = 10$ allowed our QCP to handle DAGs with as many as one hundred subtasks while still giving near-optimal reductions in utilization. For larger DAGs, $K = 1$ gave sub-optimal but still impressive results given the size of the problem considered. The effects of different K values are discussed more in Sec. IV.

Implementation. We implemented[3] the QCP using Gurobi [25], a commercial mathematical optimization solver. A free, full-featured academic license is available.

IV. Simulated DAG Experiments

In this section, we present our synthetic task experiments and results. We simulated over 70 thousand DAGs and almost four million subtasks across nearly one thousand scenarios. For each scenario, we test the ability of our QCP to reduce per-DAG utilization and, when combined with Alg. 1, to reduce the cores required per DAG.

A. Experimental Setup for Single-DAG Systems

Each scenario is defined by the number of subtasks $|V|$ per DAG, possible costs per subtask, a model for SMT behavior, a model for precedence constraints, and K. 100 random DAGs were generated per scenario.

Subtask count. For each scenario, we selected $|V|$ from $\{10, 20, 40, 80, 100, 250, 500\}$. The scenario is *large* when $|V| \geq 100$, and *small* otherwise. Small DAG scenarios were run in combination with all other parameters. Large DAG scenarios, being very time consuming, were not.

Baseline costs. To determine the costs of individual subtasks without SMT, each scenario had costs selected from a *narrow* $(1-2)$ or *wide* $(1-20)$ uniform distribution. Our expectation was that SMT would provide a greater benefit with the narrow range, as our model prohibits pairs where c_i and c_j differ by an order of magnitude, and pairing tasks with very different costs provides less overall utilization benefit. This expectation was confirmed.

SMT behavior. To generate realistic paired costs, we follow Osborne and Anderson [41] by defining a *multithreading score*, a random variable which determines $c_{i(j)}$ given c_i and c_j, assuming that $c_i \geq c_j$ holds. We repeat that definition here, adapted for our terminology.

Def. 9. [41] If $v_{i:j}$ is a subtask pair for which $c_i \geq c_j$ holds, then the *multithreading score* $m_{i:j}$ satisfies the following:

$$c_{i(j)} = c_i + m_{i:j} \cdot c_j. \blacktriangleleft \quad (2)$$

Essentially, $m_{i:j}$ dictates how much v_i's execution time increases relative to c_j. This reflects that if c_j is much less than c_i, than its effect on v_i should also be small. If $m_{i:j} \geq 1$ holds, then there is no benefit to pairing v_i and v_j together. If $m_{i:j} < 1$ holds, then pairing jobs of the two tasks is potentially beneficial, with lower values indicating greater benefit. $m_{i:j} = 0$ indicates that v_j has no effect on v_i's execution time; essentially, v_j executes for free.

Def. 9 says nothing about the behavior of the shorter task within each pair. To model the shorter task's cost, we expand the definition of $m_{i:j}$ to supersede Def. 9.

Def. 10. For all subtask pairs $v_{i:j}$ the *multithreading score* $m_{i:j}$ satisfies the following:

$$c_{i(j)} = \max\left(c_i + m_{i:j} \cdot \min(c_i, c_j), c_{j(i)}\right).$$

There is no expectation that $m_{i:j} = m_{j:i}$ will hold; this reflects the fact that SMT interactions are more complex than a simple formula. $\blacktriangleleft$

For $c_i \geq c_j$, this expression is equivalent to Exp. 2. In either case, the increase in runtime is proportional to the shorter task. The max function guarantees that the initially shorter subtask will not become the longer subtask. This model is consistent with behavior observed in [39].

We consider three possible distributions for $m_{i:j}$. For each distribution, the first number gives the probability of the pair having an arbitrarily large cost with SMT. Apart from that possibility, the multithreading scores for the pair are drawn from the given Normal distribution with mean and standard deviation $N(\mu, \sigma)$.

The distributions are *optimistic* $[0; N(0.34, 0.2)]$; *mid-range* $[0.05; N(0.52, 0.17)]$; and *pessimistic* $[0.2; N(0.6, 0.07)]$. The optimistic and mid-range distributions are based on observations by Bakita et al. [4]

[3]Code available online: https://jamesanderson.web.unc.edu/papers/ and https://github.com/shosborn/SMT-DAGs.

of the Data-Intensive Systems (DIS) [16] and San Diego Vision [49] benchmarks, respectively. The pessimistic distribution is based on Osborne and Anderson's [41] analysis of the TACLe sequential benchmarks [19]. In all cases, negative values are replaced by 0.01.

Precedence constraints. After assigning costs to all subtasks within our DAG, we use one of two methods to determine precedence constraints within our DAG.

In the *Erdős-Rényi* method, every pair of subtasks have a probability p of being connected by an edge [15, 20]. Lower values of p will produce DAGs with more potential for parallelism. We use the p values $\{0.1, 0.3, 0.5\}$. In preliminary experiments, we found that DAGs with $p > 0.5$ had very few subtasks executable in parallel and thus received little to no benefit from SMT.

In the *layer-by-layer* method, each DAG is first divided into ℓ layers. No precedence constraints ever exist between subtasks within the same layer, but subtasks in different layers have probability p of being connected by an edge [15, 46]. We use the p values $\{0.1, 0.3, 0.5, 0.7, 0.9, 1\}$. While $p = 1$ specifies a task with no parallelism in the Erdős-Rényi method, in this case it means that all subtasks in one layer must complete before the next layer can begin.

For ℓ, we use the values $\{2, 4, 8, 16\}$, with the caveat that $\frac{|V|}{\ell} \geq 5$ must hold, i.e. the expected number of subtasks per layer must be at least five. To divide the subtasks into layers, we randomly select $\ell - 1$ integers from the range $[1, |V|]$ without replacement. These integers subdivide the set of tasks into layers by index.

Ex. 5. Suppose we wish to build a DAG of 20 subtasks in three layers. Two values are chosen from the range $[1, 20]$. If we select 3 and 10, the first layer consists of subtasks v_1 through v_3, the second of subtasks v_4 through v_{10}, and the third layer of all remaining subtasks. ◀ .

This method was designed to make the DAG's total length L easily controllable; no chain in a DAG created this way will have more than ℓ members [15, 46]. Setting $\ell = |V|$ is equivalent to using the Erdős-Rényi method.

Tunable parameter K. Our final parameter is the value of K as defined in restriction (vi) of the QCP. We use K values from the set $\{1, 10, 20, 40\}$ except in large scenarios, where only $K \in \{1, 10\}$ was feasible. The QCP is only optimal when restriction (vi) is removed by setting $K \geq |V|$, but we observed excellent results for all $K \geq 10$.

DAG creation. Each DAG's total utilization is selected uniformly from the range $(1, \frac{C}{L})$. Doing so implicitly determines period and deadline (as we assume implicit deadlines, i.e. $D = T$). We do not consider DAGs with $U < 1$ as those need not be scheduled in parallel. DAGs for which $U > \frac{C}{L}$ (equivalently, $D < L$) holds are not schedulable on any number of cores, with or without SMT. Note that $\frac{C}{L}$ is itself a function of individual DAG structures. Consequently, the distribution of utilization across all DAGs in a scenario will not necesarilly be uniform.

B. Single DAG Results

We evaluated the QCP's effect on each scenario using three metrics: *relative core count*, *core reduction frequency*, and relative utilization.

Def. 11. *Relative core count* (respectively, *relative utilization*) is defined as a DAG's required core count, per Alg. 1 (respectively, total utilization) without SMT divided by the same values after applying SMT. They are abbreviated as *RCC* and *RU*. ◀

Values of one indicate no change due to SMT; values less than one indicate SMT has reduced the DAG's utilization or core count requirement.

Def. 12. *Core reduction frequency* is defined as the number of cases where the required core count was reduced by at least one divided by the total number of cases evaluated. It is abbreviated *CRF*. ◀

Note that while smaller values are better for *RCC* and *RU* larger values are better for *CRF*.

In rare cases—less than 1% of all DAGs created—we found that applying SMT would *increase* the number of cores needed. In these cases, the best choice is to not use SMT, so we set *RU* and *RCC* to one and *CRF* to zero.

We summarize each scenario with a scatterplot that plots SMT utilization, baseline cores, and SMT cores on the vertical axis against baseline utilization on the horizontal axis. In the majority of scenarios, SMT utilization appears to be a linear function of baseline utilization; in these cases, *RU* approximates the function's slope.

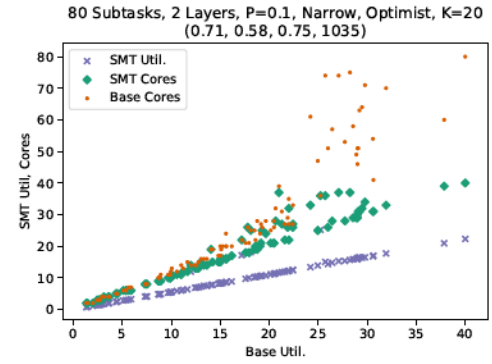
We classify scenarios as either *good*, *moderate*, or *poor* based on mean *RCC*. $RCC < 0.8$ is good; $RCC > 0.9$ is poor; and intermediate *RCC* values are moderate.

For $K \geq 10$, we have 126 good scenarios, 133 moderate scenarios, and 115 poor scenarios. For $K = 1$, we had 41 good, 161 moderate, and 222 poor scenarios. Mean *RCC* values per scenario ranged from 0.73 to 1. We saw $RU = 1$ in 29 out of nearly one thousand scenarios; of those, only four had $K \geq 10$. Mean *RU* values ranged from 0.57 to 1, and *CRF* values from 0.8 to 0.

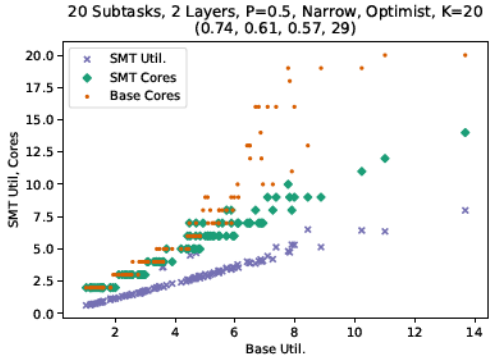
QCP execution times. We also tracked solver runtimes for each scenario. We executed the QCP on a shared cluster, so our recorded runtimes should be considered mainly for order-of-magnitude comparisons. Recorded times are for serial execution; we used parallelism to analyze many DAGs at once, but not to reduce the analysis time for individual DAGs. Our per-DAG runtimes ranged from less than a second to nearly two days. The total CPU time for analyzing all DAGs was roughly four years.

Our full set of graphs, along with code and a .csv file summarizing all results, is available online.[4] Here

[4]https://jamesanderson.web.unc.edu/papers/

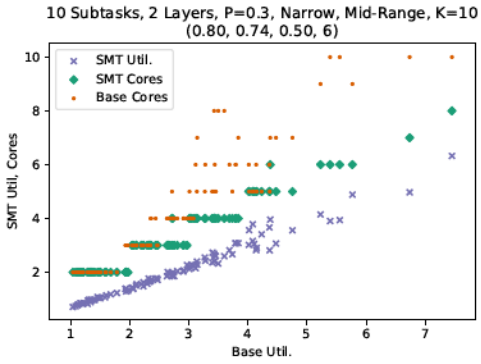


(a)

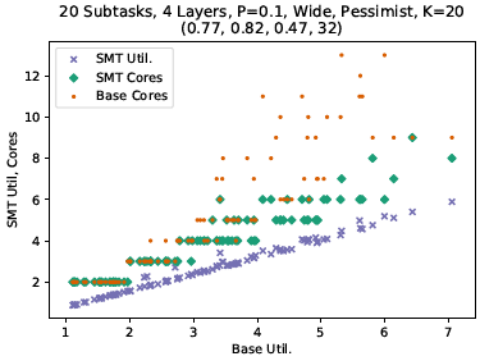


(b)

Fig. 4. Graphs for the good scenarios.



(a)



(b)

Fig. 5. Graphs for the moderate scenarios.

we show graphs that are representative of our best and worst results or that demonstrate particularly interesting behaviors. Each graph's title identifies its scenario; if the number of layers is unstated, then the graph was created using the Erdős-Rényi method. In addition, graph titles summarize performance with the tuple (mean *RCC*, mean *RU*, *CRF*, mean runtime in seconds).

Small DAG scenarios. We first consider scenarios where $|V| \leq 80$ holds. Scenarios with larger subtask counts are considered separately further down.

We show graphs for two scenarios each to represent good, moderate, and poor scenarios.

Obs. 1. Utilization is consistently decreased, even when $K < |V|$ holds. In the best cases, relative utilization is as little as 58%, as seen in Fig 4a. The first scenario of Fig. 4 is not optimal; we use $K = 20$, which is less than the subtask count of 80.

Obs. 2. The greatest benefits to core count come from avoiding cases where many DAG subtasks require a dedicated core to execute. This effect can be seen most clearly in Figs. 4 and 5a; notice that the plotted points for base cores show core requirements approaching the total subtasks per DAG.

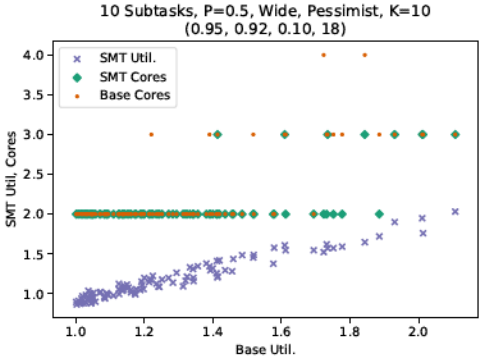
The reason is as follows: without SMT, two subtasks with combined $U > 1$ cannot be scheduled on a single core; with SMT, scheduling the two together becomes possible if the corresponding pair has $U \leq 1$. In a DAG that consisted entirely of subtasks with $U = 0.51$, SMT might be able to halve the required core count.

Obs. 3. In some cases, such as Fig. 5b, applying SMT reduces core count by more than it reduces utilization. This phenomenon may occur when small reductions in utilization enable a DAG to be scheduled on fewer cores. For example, a DAG with $U = 1.01$ requires two cores under federated scheduling, but reducing its utilization to 0.99 would allow to to execute on a single core.

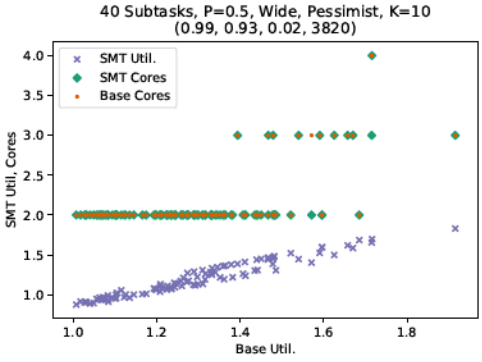
Obs. 4. The greatest improvements in terms of RU were seen with the following scenario parameters: optimistic SMT distribution; $\min(K, |V|) \geq 20$; and the layer-by-layer method was used or $p \leq 0.3$ held. All scenarios fitting these criteria had $RU < 0.8$.

The role of SMT distribution in observed improvements should be obvious. Having relatively large values for K and $|V|$ ensures that for each subtask, there are many possible ways it can be paired, making it more likely that a good pairing can be found.

Using either the layer-by-layer approach or a small p value in the Erdős-Rényi method also ensures that each subtask will have many possibilities for pairing; recall that



(a)



(b)

Fig. 6. Graphs for the poor scenarios.

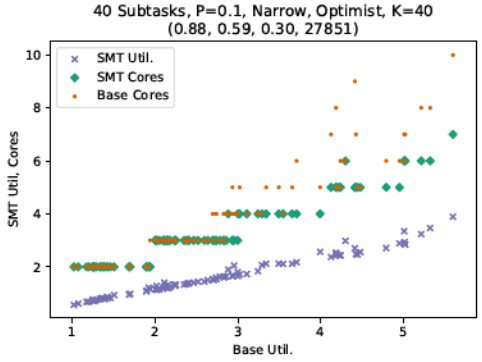
in the layer-by-layer method, a subtask can, at a minimum, be paired with others in the same layer.

Obs. 5. The smallest improvements to RU (excluding cases where $K = 1$; these are discussed in Obs. 6 below) were seen when using the pessimistic SMT distribution, the wide utilization distribution, $\min(K, |V|) < 20$) held, and graphs were constructed using the Erdős-Rényi method with $p > 0.3$. For example, see Fig. 6a. These factors, not surprisingly, are the opposite of those listed in Obs. 4.

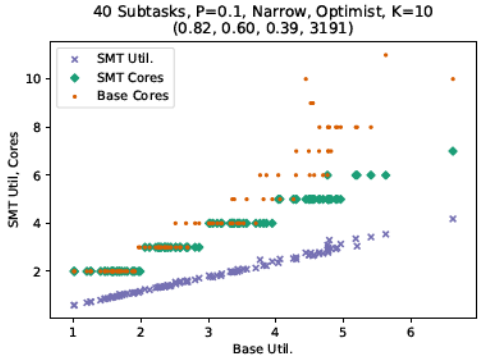
Obs. 6. Decreasing K from $|V|$ to 10 had little effect on RCC, RU, or CRF but dramatically reduced runtime. Setting $K = 1$ is more detrimental, but is better than not using SMT at all. Fig. 7 gives an example. Decreasing K from 40 in Fig. 7a to 10 in Fig. 7b has little effect on RU or RCC but reduces computation time by a factor of 10. Setting $K = 1$ in Fig. 7c increases RU from 0.60 to 0.75 and reduces computation time by another factor of 10.

Large DAG scenarios. Applying SMT to DAGs with 100 or more subtasks was difficult. We saw scenarios where every DAG required hours or days to analyze.

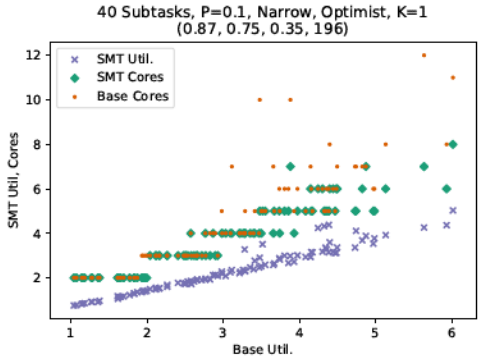
Even so, the K parameter allowed us to benefit from SMT in many large DAG scenarios. For $|V| = 100$, we tested scenarios with $K = 10$ and $K = 1$. For $|V| \in \{250, 500\}$, our QCP worked well only when $K = 1$; for



(a)



(b)



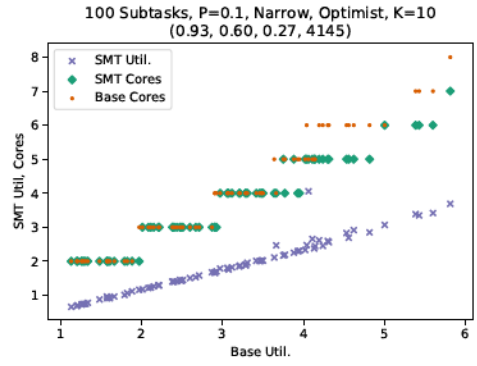
(c)

Fig. 7. Scenarios with different K values, otherwise identical parameters.

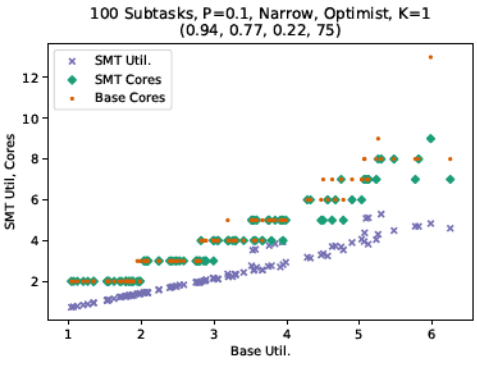
larger K values applying SMT to even a single DAG took over a day. We therefore limited these DAGs to $K = 1$.

Obs. 7. When $K = 10$ held, results were comparable to those for similar parameters for smaller $|V|$ and larger K values. For example, compare Fig. 8a to Figs. 7a and 7b.

Obs. 8. When $K = 1$ held but scenario parameters were otherwise favorable, RU values per scenario ranged from 0.70 to 0.90. Examples can be seen in Figs. 8b and 9. Note that apart from the $|V|$ and K parameters, Figs. 8 and 9 depict identical scenarios.

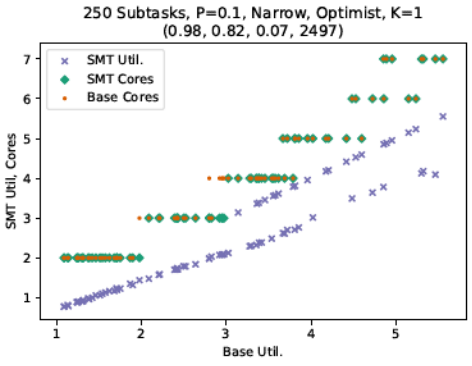


(a)

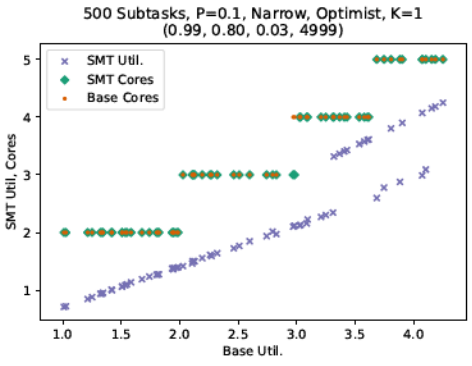


(b)

Fig. 8. 100 subtasks with $K = 10$ and $K = 1$.



(a)



(b)

Fig. 9. Scenarios with 250 and 500 subtasks.

C. Multiple DAGs and Federated Scheduling

In this subsection, we consider the case of scheduling multiple DAGs simultaneously using federated scheduling.

Creating systems of DAGs. Within each system, all DAGs are created using a single set of parameters (discussed below). The number of DAGs per system ranged from 1 to $32/E(U_i)$, where $E(U_i)$ gives the expected value of a single DAG's baseline utilization; essentially we created systems with total utilization up to approximately 32. For each step from 1 to $32/E(U_i)$, we created 10 systems. A scenario consists of all systems built using a single set of per-DAG parameters.

Per-DAG parameters. We defined four categories of per-DAG utilization: *light, low-heavy, medium-heavy,* and *high-heavy*. Total utilizations within each category are randomly chosen from the uniform ranges (0, 1], (1, 2], (2, 4], and (4, 8] respectively.

Light DAGs can all be scheduled sequentially on a single processor, even without SMT. Within the context of federated scheduling, SMT may allow more light DAGs to be scheduled per processor.

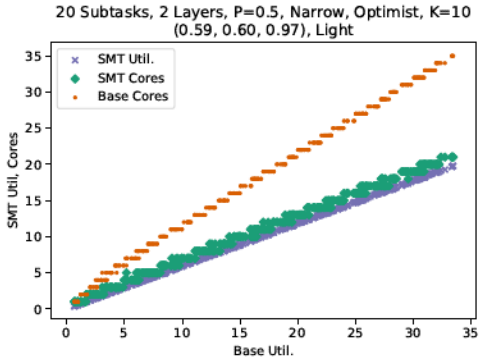
All other DAGs, since they have $U > 1$, are considered heavy per Def. 3 and would, without SMT, require at least two dedicated cores each. Our expectation is that SMT will convert many of the low-heavy DAGs into light DAGs with $U \leq 1$. Medium-heavy and high-heavy DAGs will generally remain heavy even after SMT has been applied.

In addition to the four categories of per-DAG utilization, we used the parameters that were already discussed in Sec. IV-A. We considered DAGs consisting of 10, 20, or 40 subtasks each, both the Erdős-Rényi method with $p \in \{0.1, 0.5\}$ and the layer-by-layer method with $\ell \in \{2, 4\}$ and $p \in \{0.1, 0.5, 0.9\}$, the Narrow and Wide cost ranges, and the Optimistic and Pessimistic ranges. We used $K = 10$ in all cases. Due to the time-consuming nature of these scenarios—each scenario includes 50 to almost 500 individual DAGs—we did not create scenarios for all combinations of parameters.

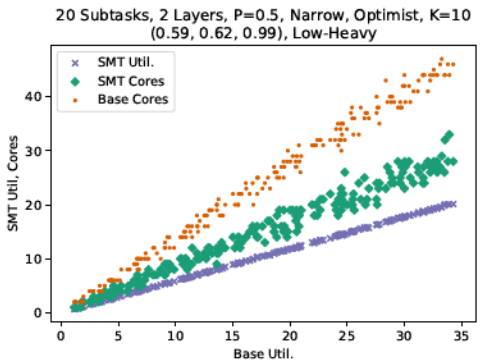
D. Federated Scheduling Evaluation

As with individual DAGs, we evaluate each scenario on the basis of *RCC*, *RU*, and *CRF*. Scenario results are shown with scatterplots similar to those for single-DAG scenarios. This time, utilization and cores needed are totals for all DAGs in each system.

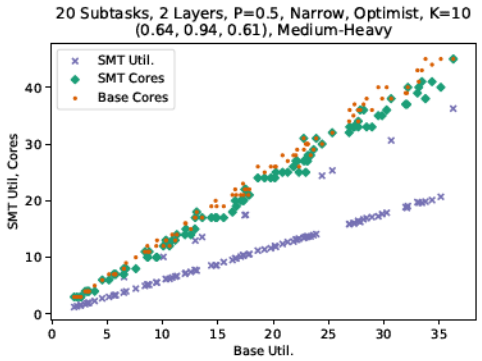
For each heavy DAG, the cores required are again determined by Alg. 1. For light tasks, we used partitioned scheduling, i.e. each task is assigned to a single core, but each core can contain multiple tasks up to total utilization 1. Tasks are assigned to cores using a bin-packing based



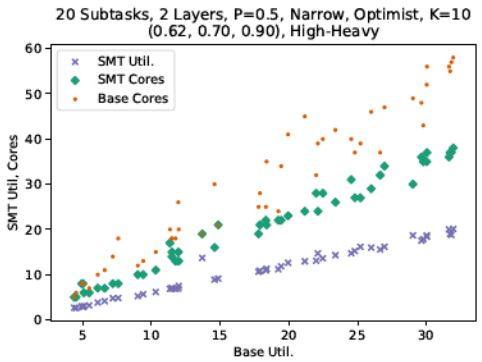
(a) light



(b) low-heavy

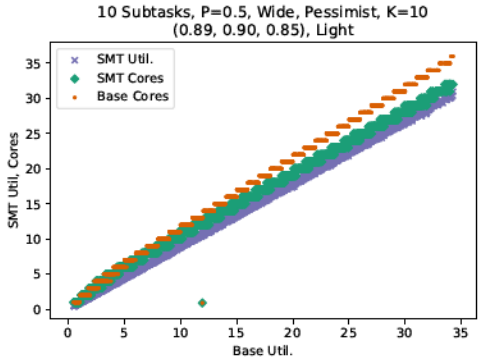


(c) medium-heavy

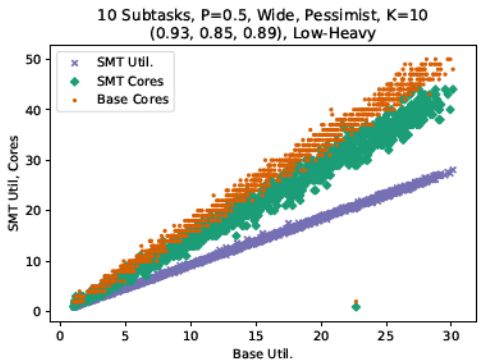


(d) high-heavy

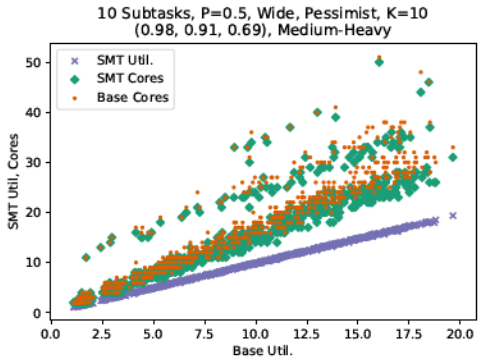
Fig. 10. Good scenarios for Federated Scheduling.



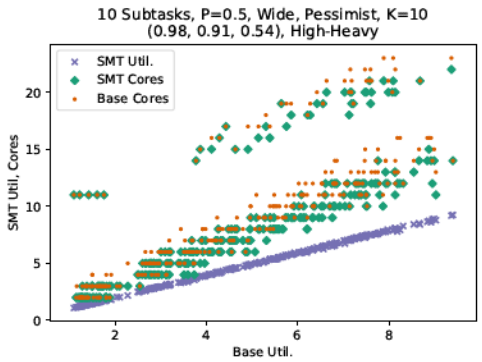
(a) light



(b) low-heavy



(c) medium-heavy



(d) high-heavy

Fig. 11. Bad scenarios for Federated Scheduling.

approach. For each system, we attempted both decreasing-best-fit and decreasing-worst-fit and used the result that required the fewest cores.

We tested 116 scenarios. Of these, using the same criteria as before, 51 were good, 36 moderate, and 28 poor.

We include the scatterplots for 8 federated scheduling scenarios. Fig. 10 depicts four good scenarios and Fig. 11 four poor scenarios. The scenarios of each figure are identical apart from the per-DAG utilizations. Note that Fig. 4b depicts the single-DAG version of the scenarios in Fig. 10 and Fig. 6b gives the single-DAG scenario corresponding to those of Fig. 11. Again, the full set of graphs is included in an online appendix.

Obs. 9. The poorest results are generally seen with per-DAG utilizations in the medium-heavy range ($U(2, 4]$).

The other three utilization ranges are well-situated to take advantage of SMT: SMT allows two light tasks with combined $U > 1$ to share a core; low-heavy tasks will likely have $U > 1$ after SMT is applied, making them light, and high-heavy tasks are large enough that utilization decreases will generally translate to core count decreases.

Obs. 10. The advantages of SMT are generally greater with federated scheduling than in the single-DAG cases.

Compared to the corresponding single-DAG scenario, each federated scenario generally performed slightly better in terms of *RCC*, about the same in terms of *RU*, and much better in terms of *CRF*. This can be observed by comparing Figs. 10 and 11 with Figs. 4b and 6b, respectively. The most dramatic example is in the change in *CRF* for the bad cases: in Fig. 6b, we have *CRF* of 0.10, but the minimum *CRF* for the Fig. 11 examples is 0.54.

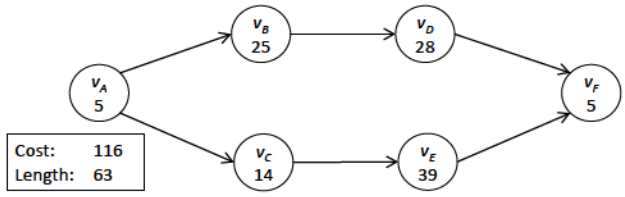
V. Case Study

In this section, we describe our case study and summarize its results. Our goal is to demonstrate that schedules generated using our methods are safe and implementable on a real system.

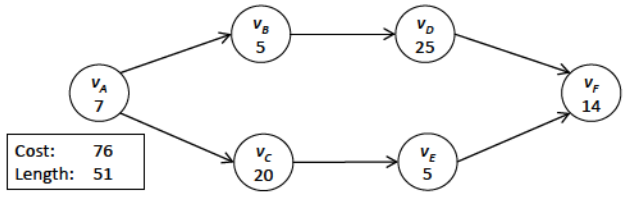
Hardware platform. Our experiments were performed on the AMD Ryzen 3950X. The 3950X includes three cache levels. The L3 cache is divided into four partitions of 16MB each, each of which is shared among four cores. Each core has a 512KB L2 cache and separate 32KB L1 caches for instructions and data.

We assigned individual hardware threads their own slices of L3 and L2 cache using the techniques of Bakita et al. [4]. This eliminates cross-thread L2 and L3 cache interference, thus improving SMT timing predictability.

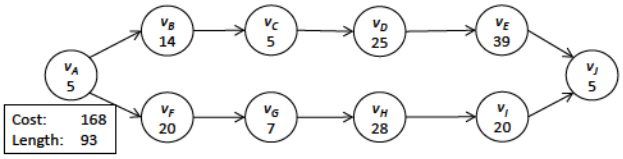
Case study DAGs. Our case study DAGs are depicted in Fig. 12. Each DAG is built by defining precedence constraints between elements of the San Diego Vision Benchmark Suite [49] using the "qcif" input option; these same benchmarks are the basis of the mid-range SMT



(a) DAG task 1.



(b) DAG task 2.



(c) DAG task 3.

Fig. 12. Case study DAG tasks (1.1x safety factor) before applying SMT.

TABLE I
The SD-VBS benchmark used for each node of each of the three case study DAGs.

Node	DAG 1	DAG 2	DAG 3
v_A	mser	tracking	texture
v_B	localization	mser	disparity
v_C	disparity	stitch	mser
v_D	svm	localization	localization
v_E	sift	texture	sift
v_F	texture	disparity	stitch
v_G	N/A	N/A	tracking
v_H	N/A	N/A	svm
v_I	N/A	N/A	stitch
v_J	N/A	N/A	texture

distribution for our synthetic tasks. Table I lists which SD-VBS benchmark is run for each node in the three DAGs. By building these "semi-synthetic" DAGs from established benchmarks, we obtained DAGs that are reflective of common DAG structures, allow us to highlight key possibilities for SMT, and are small enough to be easily understood.

Subtask costs. To apply SMT to our DAGs, we needed to know all execution times both with and without SMT. We obtained this data previously in [4], where we measured 1000 runs of each element of the benchmark suite both without SMT and paired with every other element. We used two sets of costs: the obtained maximums multiplied

TABLE II
SCHEDULE FOR DAG 3 (FIGURE 12C) WITH SMT, $D = 121$MS. START TIMES AND COSTS IN MS. USING 1.1X SAFETY FACTOR.

Subtask 1	Subtask 2	Core	Start	Costs
A	N/A	0	0	5
B	F	0	5	21; 27
C	N/A	1	26	5
G	N/A	0	32	7
D	H	0	39	36; 39
E	N/A	1	75	39
I	N/A	0	78	20
J	N/A	0	114	5

by a safety margin of 1.5 and by a safety margin of 1.1. All costs were computed and all tests were run using LITMUS-RT, a real-time extension of the Linux kernel [7, 11, 35]. We also disabled power management and used Linux's `isolcpus` and `irqaffinity` options to minimize interference during our case study.

Building schedules. For each of our three DAGs, multiple deadlines—two for DAG 2 and three each for DAGs 1 and 3—were used to provide either no, some, or unlimited opportunity to use SMT. The case study thus covers DAG tasks where the application of SMT may be not beneficial, somewhat beneficial, or very beneficial.

For each of the eight DAG-deadline combinations with each set of costs, we applied the QCP of Sec. III with $K = |V|$ and Alg. 1 to generate a schedule. We then implemented each schedule using LITMUS-RT [7, 11, 35]. Each subtask was assigned a pseudo-deadline equal to its start time plus its cost.

Table II shows the schedule generated for DAG 3 (Fig. 12c) using the 1.1x safety factor with a deadline of 121ms (moderate length), including the subtask pairings, core assignments, start times, and costs. For paired subtasks, Subtask 1 has the shorter paired cost. Note that, as in Ex. 4, a subtask can begin as soon as all predecessors are complete, including predecessors that are paired with a longer-running task. Here, subtask C begins as soon as B completes, it does not wait on F, the partner of B.

We found that paired subtasks did in fact begin virtually simultaneously, as required by Def. 4 and restriction (ii). Among all paired subtasks in all DAGs we ran, the largest difference between "simultaneous" start times we observed was $870ns$, with a mean of $80ns$ and median of $40ns$. In addition, we calculated the mean and median differences over all ten thousand iterations for each pair, and find the maximum of the means is $128ns$ and the maximum of the medians is $90ns$. Compared to the execution times of our subtasks—all in the ms range—these differences are insignificant, showing that our paired subtasks are as close to starting simultaneously as is realistically possible.

Results. For each DAG-deadline and cost combination, we ran ten thousand DAG-jobs over multiple hours. No DAG deadlines or subtask pseudo-deadlines were missed. This result supports our claim that timing analysis with SMT is reliable enough to schedule DAGs on real hardware when the safety conditions are met.

In addition to testing each of our DAGs individually, we also experimented with running 3 DAGs at once in a federated scheduling context. The system consists of two light DAGs (DAG 1 with $D = 120ms$ and DAG 2 with $D = 80ms$) and one heavy DAG (DAG 3 with $D = 139ms$). Utilizing SMT, each DAG is able to execute on an independent core. We repeated the DAG-jobs ten thousand times while monitoring deadline misses. We observed 0 deadline misses in these experiments.

VI. CONCLUSION

To reduce the utilization of DAG tasks, we explored the possibilities of using SMT to run individual subtasks in parallel. We defined a QCP to apply SMT to DAGs that can minimize DAG utilization or to find a demonstrably excellent approximation of the best way to apply SMT.

To test our algorithm, we simulated over 70 thousand DAGs across thousands of scenarios. We found that, within our tested scenarios, SMT can reduce DAG utilization by nearly half and, at high utilization levels, can do the same for DAG core requirements. Of nearly 400 scenarios we solved near-optimally—i.e. $K \geq 10$ held—40% had their utilization reduced by at least 25%. These near-optimal results include DAGs with as large as 100 subtasks. Given favorable scenarios, we were able to see similar utilization reductions for DAGs with as many as 500 subtasks.

In addition to evaluating the effects of SMT on individual DAGs, we also applied our methods to systems of DAGs using federated scheduling. We found that systems of DAGs generally saw greater improvement than did individual DAGs with similar per-DAG parameters.

To further support our findings, we performed a case study in which we applied SMT to real DAG workloads and implemented the resulting schedule. All of our DAGs executed perfectly, with no deadline misses.

In the future, we hope to expand our work to include more complex models which could not fully fit within the body of this paper. We also hope to supplement our synthetic-task based analysis with a more theoretical approach.

ACKNOWLEDGEMENT

We wish to our anonymous reviewers and shepherd for their helpful contributions, particularly with regards to the federated scheduling discussion in Sec. IV.

REFERENCES

[1] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis. Outstanding paper: Evaluation of cache partitioning for hard real-time systems. In *ECRTS 2014*, pages 15–26. IEEE, 2014.

[2] T. Amert, M. Balszun, M. Geier, F. D. Smith, J. Anderson, and Samarjit C. Timing-predictable vision processing for autonomous systems. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1739–1744. IEEE, 2021.

[3] J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, volume 4, page 7. Euromicro, 2009.

[4] J. Bakita, S. Osborne, S. Ahmed, S. Tang, J. Chen, F.D. Smith, and J. Anderson. Simultaneous multithreading in mixed-criticality real-time systems. In *RTAS*, 2021.

[5] S. Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1323–1328. IEEE, 2015.

[6] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS 2012*, pages 63–72. IEEE, 2012.

[7] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems.* PhD thesis, UNC Chapel Hill, 2011. https://www.cs.unc.edu/ anderson/diss/bbbdiss.pdf.

[8] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA 2008*, pages 101–110. IEEE, 2008.

[9] J. Bulpin. *Operating system support for simultaneous multithreaded processors.* PhD thesis, University of Cambridge, King's College, 2005. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-619.pdf.

[10] J. Bulpin and I. Pratt. Multiprogramming performance of the Pentium 4 with hyperthreading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, 2004.

[11] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. Anderson. LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*, pages 111–126, 2006.

[12] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, July 2006.

[13] M. Chisholm, N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter. Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *RTNS 2017*, pages 58–67. ACM, 2017.

[14] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS 2015*, pages 305–316. IEEE, 2015.

[15] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random graph generation for scheduling simulations. In *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*, page 10. ICST, 2010.

[16] Atlantic Aerospace Division. DIS Stressmark Suite. Technical report, Titan Systems Corporation, 2000. ver. 1.0.

[17] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Sept 1997.

[18] G.. Elliott, K. Yang, and J. Anderson. Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *RTSS 2015*, pages 273–284, 2015.

[19] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *WCET*, 2016.

[20] E. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[21] T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares. Bringing hardware multithreading to the real-time domain. *IEEE Embedded Systems Letters*, 8(1):2–5, March 2016.

[22] T. Gomes, S. Pinto, P. Garcia, and A. Tavares. RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *ETFA*, 2015.

[23] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[24] D. Guo and R. Pellizzoni. A requests bundling DRAM controller for mixed-criticality systems. In *RTAS 2017*, pages 247–258. IEEE, 2017.

[25] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[26] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS 2015*, pages 307–316. IEEE, 2015.

[27] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *RTAS 2012*, pages 197–208, April 2012.

[28] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS*, 2002.

[29] X. Jiang, N. Guan, and W. Long, X.and Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *RTSS 2017*, pages 80–91. IEEE, 2017.

[30] H. Kim, J.and Kim, K. Lakshmanan, and Ragunathan (Raj) Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, page 31–40. ACM, 2013.

[31] J. Kim, M. Yoon, R. Bradford, and L. Sha. Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 321–331, July 2014.

[32] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *RTSS 1989*, pages 229–237. IEEE, 1989.

[33] J. Li, J. J. Chen, K.l Agrawal, C. Lu, C.s Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *ECRTS 2014*, 2014.

[34] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu. Randomized work stealing for large scale soft real-time systems. In *RTSS 2016*, pages 203–214, 2016.

[35] LITMUS^{RT} home page. Online at http://www.litmus-rt.org/, 2020.

[36] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS 2013*, pages 45–54, April 2013.

[37] A. Melani, R. Mancuso, M. Caccamo, G. Buttazzo, J. Freitag, and S. Uhrig. A scheduling framework for handling integrated modular avionic systems on multicore platforms. In *RTCSA 2017*, pages 1–10, 2017.

[38] D. Muench, M. Paulitsch, and A. Herkersdorf. Temporal separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using PCIe SR-IOV. In *2014 Workshop Proceedings on Architecture of Computing Systems (ARCS)*, pages 1–7, Feb 2014.

[39] S. Osborne, S. Ahmed, S. Nandi, and J. Anderson. Exploiting simultaneous multithreading in priority-driven hard real-time systems. In *RTCSA*, 2020.

[40] S. Osborne and J. Anderson. Work in progress: Combining real time and multithreading. In *RTSS 2019*, pages 139–142, 2018.

[41] S. Osborne and J. Anderson. Simultaneous multithreading and hard real time: Can it be safe? In *ECRTS*, 2020.

[42] S. Osborne, J. Bakita, and J. Anderson. Simultaneous multithreading applied to real time. In *ECRTS 2019*, 2019.

[43] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *RTSS*, 2008.

[44] G. N. Seetanadi, J. Camara, L. Almeida, K. Arzen, and M. Maggio. Event-driven bandwidth allocation with formal guarantees for camera networks. In *RTSS 2017*, pages 243–254, Dec 2017.

[45] K. Suito, K. Fujii, H. Matsutani, and N. Yamasaki. Dependable responsive multithreaded processor for distributed real-time systems. In *IEEE COOL Chips XV*, 2021.

[46] T. Tobita and H.i Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[47] N. Ueter, G. Von Der Brüggen, J.-J. Chen, J. Li, and K. Agrawal. Reservation-based federated scheduling for parallel real-time tasks. In *RTSS 2018*, pages 482–494. IEEE, 2018.

[48] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS 2016*, pages 1–12, April 2016.

[49] S. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. Taylor. SD-VBS: the San Diego vision benchmark suite. In *IISWC*, pages 55–64, 2009.

[50] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS 2007*, pages 239–243, 2007.

[51] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *ECRTS 2013*, pages 157–167, July 2013.

[52] M. Xu, L. T. X. Phan, H. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS 2016*, pages 1–12, April 2016.

[53] K. Yang, G. Elliott, and J. Anderson. Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *RTNS 15*, page 77–86. ACM, 2015.

[54] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, and J.-M. Anderson, J.and Frahm. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *RTAS 2019*, pages 305–317. IEEE, 2019.

[55] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS 2014*, pages 155–166, April 2014.

[56] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, pages 55–64, April 2013.

[57] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *RTAS 2014*, April 2014.