Assessing the Computational Benefits of AREA-Oriented DAG-Scheduling

Gennaro Cordasco, Rosario De Chiara
Università degli Studi Salerno, Italy
Email: {`cordasco,dechiara`}`@dia.unisa.it`

Arnold L. Rosenberg
Colorado State University, USA
Email: `rsnbrg@cs.umass.edu`

***Abstract*—Many modern computational platforms, including "aggressive" multicore architectures, proposed exascale architectures, and many modalities of Internet-based computing are "task hungry"—their performance is enhanced by always having as many tasks eligible for allocation to processors as possible. The *IC-scheduling* paradigm for computations with inter-task dependencies—modeled as DAGs—was developed to address the "hunger" of such platforms, by executing an input DAG so as to render tasks eligible for execution as fast as possible. The fact that many DAGs do not admit schedules that are optimal under IC-scheduling spawned the development of a new paradigm—*AREA-Oriented scheduling* (AO-scheduling)—that coincides with optimal IC-scheduling on DAGs that admit IC-optimal schedules but that allows optimal AO-scheduling *of all* DAGs. AO-scheduling achieves its universal applicability by weakening the often-unachievable demand of IC-scheduling that the number of eligible tasks be maximized at *every* step when executing a DAG to the always-achievable demand that this number be maximized *on average*. The computational complexity of optimal AO-scheduling is not yet known; therefore, this goal is replaced here by a multi-phase heuristic that produces optimal AO-schedules for series-parallel DAGs but possibly suboptimal schedules for general DAGs.**

As with IC-scheduling, it is not clear *a priori* that AO-scheduling enhances the efficiency of executing a DAG by minimizing the makespan of its execution. This paper employs simulation experiments to assess the computational benefits of AO-scheduling in a variety of scenarios and on a range of DAGs whose structure is reminiscent of ones encountered in scientific computing. The experiments pit AO-scheduling against a variety of heuristics that range from lightweight ones such as FIFO scheduling to computationally more intensive ones that mimic IC-scheduling's *local* decisions. The observed results indicate that, statistically, AO-scheduling does enhance the efficiency of task-hungry platforms, by amounts that vary according to the availability patterns of processors and the structure of the DAG being executed.

***Keywords*-Scheduling DAGs; Scheduling for: task-hungry platforms, multicore architectures, exascale architectures**

I. Introduction

Many modern computational platforms, including "aggressive" multicore architectures (cf. [29]), proposed exascale architectures (cf. [12]), and many modalities of Internet-based computing (cf. [15], [19], [20], [27]), are "task hungry"—their performance is enhanced by always having as many tasks eligible for allocation to processors as possible. The server-client *IC-scheduling* paradigm for computations with inter-task dependencies—modeled as DAGs—was developed to address the "hunger" of such platforms, by executing an input DAG so as to render tasks ELIGIBLE for execution as fast as possible (cf. [5], [7], [10], [23], [24], [27], [28]). Dual intuition motivates *IC-scheduling*: (1) Schedules that produce ELIGIBLE tasks/nodes more quickly may prevent a computation's stalling pending the return of already allocated tasks. (2) If the server receives many requests for tasks at (roughly) the same time, then having more ELIGIBLE tasks available allows it to satisfy more requests, thereby increasing "parallelism." The fact that many DAGs do not admit schedules that are optimal under IC-scheduling [24] spawned the development of a new paradigm—*AREA-Oriented scheduling* (AO-scheduling). Optimal AO-schedules—called *AREA-max schedules* for reasons explained in Section II—coincide with optimal IC-schedules on DAGs that admit such schedules; but, AO-scheduling allows one to develop an AREA-max schedule *for every* DAG. AO-scheduling achieves its universal optimizability by weakening the often-unachievable demand of IC-scheduling that the number of ELIGIBLE tasks be maximized at *every* step when executing a DAG to the always-achievable demand that this number be maximized *on average*. The foundations of AO-scheduling are presented for general DAGs in [8] and for *series-parallel* DAGs in [9]. Series-parallel DAGs play a central role in thread-based parallel programming, as in *Cilk* [2], [3]; they are significant in AO-scheduling because of the ease of finding AREA-max schedules for them [9]—which leads to an efficient heuristic for general AO-scheduling. The need for such a heuristic resides in results from [8] that suggest that developing AREA-max schedules for general DAGs may be computationally intractable. We respond to this possible intractability in Section III-B with a multi-phase heuristic that produces AREA-max schedules for series-parallel DAGs but possibly suboptimal AO-schedules for general DAGs. The heuristic finds an AO-schedule for a DAG $\mathcal{G}$ by:

1) using an algorithm such as those proposed in [13], [18], [25] to convert $\mathcal{G}$ to a series-parallel DAG $\mathcal{G}'$.
2) developing an AO-schedule for $\mathcal{G}$ by "filtering" the optimal AO-schedule for $\mathcal{G}'$ produced by the efficient algorithm of [9].

As with IC-scheduling, it is not clear *a priori* that AO-scheduling enhances the efficiency of executing a DAG by minimizing the makespan of its execution. The enhancement of efficiency via *IC-scheduling* is verified experimentally in [6], [16], [22] for many families of DAGs, including a broad range of randomly generated DAGs that admit IC-optimal schedules. But, as we have noted, many DAGs do not admit IC-optimal schedules—which fact motivates the current study. The current paper employs a methodology similar to that of [16] in order to assess the potential computational benefits of AO-scheduling. We model a "task-hungry" computational platform as a stream of task-seeking clients that arrive according to a random process. We focus on two random populations of DAGs.

1) We study AREA-max schedules for randomly constructed series-parallel DAGs. Such DAGs arise, e.g., via transformation from the DAGs of the preceding paragraph; cf. [13], [18], [25]
2) We study the AO-schedules produced by our multi-phase heuristic for DAGs that are random compositions of small "building-block" DAGs. (We thereby focus only on *efficiently constructed* AO-schedules.) The DAGs we schedule model computations each of whose subcomputations has the structure of *an expansion* (as in a search tree), *a reduction* (as in an accumulation), *a parallel-prefix* (a/k/a *scan*), *an all-to-all communication* (as in a "gossip"). It is shown in [6] how compositions of such DAGs represent computations such as divide-and-conquer algorithms (e.g., mergesort or numerical integration), matrix multiplication, the Fast-Fourier Transform, the Discrete Laplace Transform, and LU-decomposition. Thus, the resulting DAGs are reminiscent of ones that arise in scientific computing.

We simulate executing each generated DAG on our platform model: (a) using an AO-schedule and (b) using a variety of scheduling heuristics that range from lightweight common heuristics such as FIFO scheduling to computationally more intensive ones that mimic IC-scheduling's *local* decisions.

The results we observe indicate that, statistically, AO-scheduling *does significantly enhance the efficiency of task-hungry platforms*, by amounts that vary according to the availability patterns of processors and the structure of the DAG being executed.

II. Background

A. Basic notions

We study computations that are described by DAGs. Each DAG $\mathcal{G}$ has a set $V_{\mathcal{G}}$ of *nodes*, each representing a *task*, and a set $A_{\mathcal{G}}$ of (directed) *arcs*, each representing an intertask dependency. For arc $(u \rightarrow v) \in A_{\mathcal{G}}$:

- task v cannot be executed until task u is;
- u is a *parent* of v, and v is a *child* of u in $\mathcal{G}$.

The number of children of u is its *outdegree*. A parentless node is a *source*; a childless node is a *target*. $\mathcal{G}$ is *connected* if it is so when one ignores arc orientations. When one executes a DAG $\mathcal{G}$, a node $v \in V_{\mathcal{G}}$ becomes ELIGIBLE (for execution) only after all of its parents have been executed; hence, every source of $\mathcal{G}$ is ELIGIBLE at the beginning of an execution. The goal is to render all of $\mathcal{G}$'s targets ELIGIBLE. Informally, a *schedule* Σ for $\mathcal{G}$ is a rule for selecting which ELIGIBLE node to execute at each step of an execution of $\mathcal{G}$; formally, Σ is a *topological sort* of $\mathcal{G}$, i.e., a linearization of $V_{\mathcal{G}}$ under which all arcs point from left to right (cf. [11]). We do not allow recomputation of nodes/tasks, so a node loses its ELIGIBLE status once it is executed. In compensation, after $v \in V_{\mathcal{G}}$ has been executed, there may be new nodes that are rendered ELIGIBLE; this occurs when v is their last parent to be executed.

We henceforth refer to *tasks* rather than *nodes*, to emphasize the computational aspect of our study.

B. Series-parallel DAGs (SP-DAGs)

A *(2-terminal) series-parallel* DAG $\mathcal{G}$ (*SP*-DAG, for short) is produced via the following operations (cf. Fig. 1):

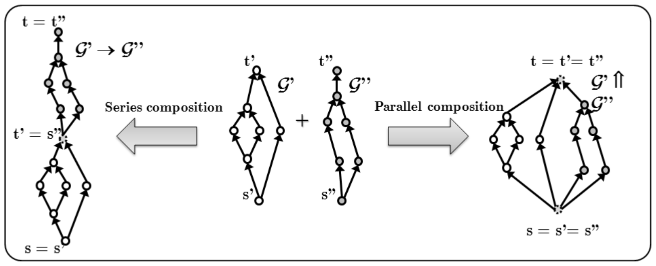

Figure 1. Compositions of SP-DAGs.

1) Create a basic SP-DAG $\mathcal{G}$, that has:
   a) two nodes, a *source* s and a *target* t, which are jointly $\mathcal{G}$'s *terminals*,
   b) one arc, $(s \rightarrow t)$, directed from s to t.
2) Compose SP-DAGs, $\mathcal{G}'$ with terminals s', t', and $\mathcal{G}''$, with terminals s'', t'':
   a) Parallel composition: Form $\mathcal{G} = \mathcal{G}' \Uparrow \mathcal{G}''$ by identifying/merging s' with s'' to form a new source s and t' with t'' to form a new target t.
   b) Series composition: Form $\mathcal{G} = (\mathcal{G}' \rightarrow \mathcal{G}'')$ by identifying/merging t' with s''. $\mathcal{G}$ has the single source s' and the single target t''.

C. Quality metrics

We measure the quality of a schedule Σ for DAG $\mathcal{G}$ via the rate at which Σ renders nodes of $\mathcal{G}$ ELIGIBLE: the faster, the better. To this end, we define $E_\Sigma(t)$, *the quality of* Σ *at step* t, as the number of nodes of $\mathcal{G}$ that are ELIGIBLE after Σ has executed t nodes[1] ($t \in [1, N_{\mathcal{G}}]$).

The *goal of IC-scheduling* is to execute $\mathcal{G}$'s nodes in an order that maximizes $E_\Sigma(t)$ *at every step* $t \in [1, N_{\mathcal{G}}]$ *of the*

[1] $[a, b]$ denotes the set of integers $\{a, a+1, \ldots, b\}$; $N_{\mathcal{G}} \overset{\text{def}}{=} |V_{\mathcal{G}}|$.

execution. A schedule Σ^* that achieves this demanding goal is *IC-optimal;* formally,

$$(\forall t \in [1, N_{\mathcal{G}}]) \quad E_{\Sigma^*}(t) \;=\; \max_{\Sigma \text{ a schedule for } \mathcal{G}} \{E_\Sigma(t)\}$$

The *goal of AO-scheduling* is to maximize the *AREA* of a schedule Σ for $\mathcal{G}$, where $AREA(\Sigma)$, is the sum

$$AREA(\Sigma) \;\stackrel{\text{def}}{=}\; E_\Sigma(0) + E_\Sigma(1) + \cdots + E_\Sigma(N_{\mathcal{G}}).$$

The normalized AREA, $\widehat{E}(\Sigma) \stackrel{\text{def}}{=} AREA(\Sigma) \div N_{\mathcal{G}}$, is the *average* number of nodes that are ELIGIBLE when Σ executes $\mathcal{G}$.[2] The goal of AO-scheduling is, thus, to find an *AREA-max schedule* for $\mathcal{G}$, i.e., a schedule $\Sigma^\star$ such that

$$AREA(\Sigma^\star) \;=\; \max_{\Sigma \text{ a schedule for } \mathcal{G}} AREA(\Sigma).$$

Easily (see [24]), many DAGs with simple structures, including many *tree*-DAG*s*[3] and *SP*-DAG*s* do not admit IC-optimal schedules. Hence, even these well-structured families benefit from the more inclusive goal of AO-scheduling.

III. Finding Good AO-Schedules Efficiently

A. The Complexity of AREA-Maximization

It is shown in [8] that, for every DAG $\mathcal{G}$:

- $\mathcal{G}$ admits an AREA-max schedule.
- If $\mathcal{G}$ admits an IC-optimal schedule, then every such schedule is AREA-max, and vice-versa.

This good news is tempered by a demonstrated close relationship between the problem of producing an AREA-max schedule for a DAG $\mathcal{G}$ and the *Maximum Linear Arrangement* (*MLA*) Problem for $\mathcal{G}$ [26]. This relationship makes it likely that the general problem of producing AREA-max schedules is computationally intractable. Fortunately, efficient such algorithms exist for two important classes of DAGs.

Lemma 3.1 ([8]): One can find an AREA-max schedule for any n-node *monotonic tree*-DAG[4] $\mathcal{G}$ in time $O(n \log n)$.

The algorithm of Lemma 3.1 adapts an algorithm from [1] that optimally solves the MLA Problem for monotonic tree-DAGs.

Lemma 3.2 ([9]): One can find an AREA-max schedule for any n-node SP-DAG $\mathcal{G}$ in time $O(n^2)$.

In brief, the algorithm of Lemma 3.2 exploits $\mathcal{G}$'s series-parallel structure in the following way.

- It decomposes $\mathcal{G}$ (using an algorithm such as the SP-DAG-recognizing algorithm of [30]) to produce the tree $\mathcal{T}_{\mathcal{G}}$ that exposes $\mathcal{G}$'s series-parallel structure; cf. Fig. 2.
- It recursively unrolls $\mathcal{T}_{\mathcal{G}}$ from the leaves up, crafting an AREA-max schedule for each of $\mathcal{T}_{\mathcal{G}}$'s node-DAGs.

[2]The term "*area*" arises by formal analogy with Riemann sums as approximations to integrals.

[3]A *tree*-DAG is a DAG that remains cycle-free when the orientation of its arcs is ignored.

[4]A tree-DAG is *monotonic* if all arcs either point away from a unique source or toward a unique target.

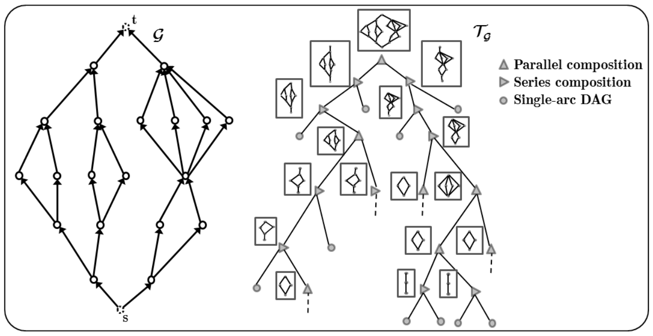

Figure 2. An example of the series-parallel decomposition of a SP-DAG.

B. Toward Efficient AO-Scheduling for General DAGs

We exploit two algorithmic sources to devise an efficient (specifically, time-$O(n^2)$) four-phase AO-scheduling heuristic for general n-node DAGs. Given an n-node DAG $\mathcal{G}$:

Phase 1: *Find $\mathcal{G}$'s transitive skeleton $\mathcal{G}'$.*

This phase, which removes all shortcut arcs from $\mathcal{G}$, reduces the overall complexity of finding the AO-schedule. Formally, $\mathcal{G}'$ is a smallest sub-DAG of $\mathcal{G}$ that shares $\mathcal{G}$'s node-set and transitive closure. Easily, $\mathcal{G}$ and $\mathcal{G}'$ share all of their AREA-max schedules, because removing shortcuts does not impact any node's dependencies.

Phase 2: *Convert $\mathcal{G}'$ to an SP-*DAG *$\sigma(\mathcal{G}')$ (SP-ize $\mathcal{G}'$).*

We invoke an *SP-ization algorithm*—i.e., an algorithm that converts an arbitrary DAG to an SP-DAG—that:

- maintains in $\sigma(\mathcal{G}')$ all of the internode dependencies inherent in $\mathcal{G}'$;
- (approximately) retains the degree of parallelism inherent in $\mathcal{G}'$ (this precludes, e.g., having $\sigma(\mathcal{G}')$ simply linearize $\mathcal{G}'$);
- operates within time $O(n^2)$.

Note that $\sigma(\mathcal{G}')$ will generally contain additional "synchronizing" nodes, which are not nodes of $\mathcal{G}'$. See Fig. 3.

One finds SP-ization algorithms that fit our requirements in sources such as [13], [18], [25]. For convenience, we use the second, more efficient, SP-ization algorithm from [13] in our experiments. It remains a topic of research to find an SP-ization algorithm that is best suited for our AO-scheduling heuristic.

Phase 3: *Find an AREA-max schedule Σ' for $\sigma(\mathcal{G}')$.*

We invoke the algorithm of Lemma 3.2. It remains a topic of research to see if a more efficient algorithm exists.

Phase 4: *"Filter" the AREA-max schedule Σ' for $\sigma(\mathcal{G}')$ to obtain the AO-schedule Σ for $\mathcal{G}$.*

(Recall that Σ' is a linearization of $\sigma(\mathcal{G}')$.) "Filtering" Σ' removes the additional nodes added by the SP-ization algorithm. For each additional node u, we assign the parents of u in $\mathcal{G}$ a *priority* that equals the priority of u in Σ'. Σ then schedules equal-priority nodes of $\mathcal{G}$ greedily, by their *yield*—the number of ELIGIBLE nodes their execution produces.

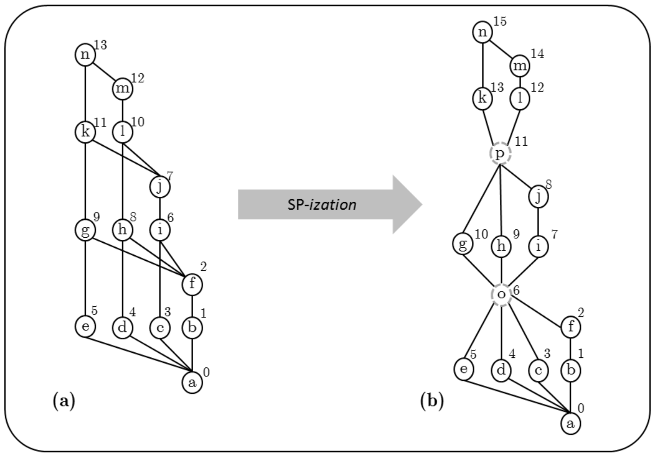

Figure 3. A sample SP-ization. Additional nodes have fuzzy borders.

We illustrate the preceding heuristic on the LU-decomposition DAG $\mathcal{G}$ of Fig. 3(a). $\mathcal{G}$ contains no shortcut arcs, so $\mathcal{G}' = \mathcal{G}$. One possible SP-ization $\sigma(\mathcal{G}')$ of $\mathcal{G}'$ appears in Fig. 3(b); note the two additional nodes, o and p. The algorithm of Lemma 3.2 produces the Area-max schedule $(a, b, f, c, d, e, o, i, j, g, h, p, l, k, m, n)$ for $\sigma(\mathcal{G}')$: note the node-numbering in Fig. 3(b). Finally, we obtain an AO-schedule Σ for $\mathcal{G}$ by simply removing nodes o and p from Σ'. We have found that the AO-schedules produced by our heuristic are more efficient if we allow them latitude in executing the parents of additional nodes—which is why we introduce the just-mentioned priority scheme. In the current case, e.g., we produce the AO-schedule $(a, b, \{c, d, e, f\}, i, \{g, h, j\}, l, k, m, n)$ and mandate that equal-priority nodes be executed greedily, by yield; this greedy strategy schedules the sets $\{c, d, e, f\}$ and $\{g, h, j\}$ in the order (c, f, d, e) and (g, j, h), respectively. By looking at the DAG $\mathcal{G}$ carefully, one can see that the final schedule, $(a, b, c, f, d, e, i, g, j, h, l, k, m, n)$, is AREA-maximizing.

IV. Our Experiments

We now describe our experiments and their results.

A. Experimental Design

1) Overview: We randomly generate DAGs from a population that shares structural characteristics with a variety of "real" computation-DAGs, especially those encountered in scientific computing. We craft five schedules for each generated DAG, one using the AO-scheduling heuristic of Section III, and four using heuristics that represent a range of sophistication and computational intensiveness. We compare the five schedules using two metrics:

1) the *batched makespan* of each schedule, which is obtained using a probabilistic model that specifies the arrival patterns of "hungry" clients and the execution time of each allocated task;
2) the *AREA* of each schedule.

Our interest in the schedules' AREAs results from the observed smaller makespans of our AO-schedules.

2) The DAGs that we execute: We generate DAGs randomly from two populations.
—*Random n-node SP-DAGs.*
—*Random n-node LEGO®-DAGs* (named for the toy). We begin, as in [24], with a repertoire of *Connected Bipartite Building Block* DAGs (CBBBs, for short). We employ various-size instances of six CBBB-structures, to represent a variety of subcomputations of the final computation-DAG.

- The leftmost two DAGs in Fig. 4 exemplify *expansive* subcomputations such as occur in a search tree.
- The third and fourth DAGs from the left in Fig. 4 exemplify *reductive* subcomputations such as occur in an accumulation tree.
- The fifth DAG from the left in Fig. 4 exemplifies pieces of subcomputations such as the parallel-prefix.
- The sixth DAG from the left in Fig. 4 exemplifies the basic building blocks of computation-DAGs such as comparators, sorters, and the Fast Fourier Transform.
- The seventh DAG from the left in Fig. 4 exemplifies the basic building blocks of computation-DAGs such as total-exchange comparators and the Fast Fourier Transform.
- The rightmost DAG in Fig. 4 exemplifies the basic building blocks of computation-DAGs that perform pivoting operations. It is also a sub-DAG of a large *functional Magnetic Resonance Imaging* DAG studied in [22].

We generate a random LEGO®-DAG by selecting a sequence of CBBBs, randomized according to both size and structure, and *composing* the CBBBs from left to right in the manner described in [24], which is depicted schematically in Fig. 5.

3) The five competing schedulers: The scheduler against which all other schedulers are measured is our AO-scheduler AO. This heuristic has two modes of operation.

1) When AO is presented with a DAG $\mathcal{G}$ that is known to be series-parallel (say, because the composition tree $\mathcal{T}_\mathcal{G}$ is provided), then AO uses the algorithm of Lemma 3.2 to craft an AREA-max schedule for $\mathcal{G}$.
2) When AO is presented with a DAG $\mathcal{G}$ that is *not* known to be series-parallel (possibly because $\mathcal{G}$ is not an SP-DAG), then AO uses the multi-phase heuristic of Section III-B to craft an AO-schedule for $\mathcal{G}$.

In either case, AO schedules an n-node DAG in time $O(n^2)$ (using our current implementation).

The heuristics that compete against AO differ in the data structures that they use to store the current ELIGIBLE tasks of $\mathcal{G}$. (The definitions and characteristics of the upcoming data structures can be found in [11].)

- The FIFO (*first-in, first-out*) scheduler organizes $\mathcal{G}$'s current ELIGIBLE nodes in a FIFO queue. It serves a "hungry" client by dequeuing the node at the front of

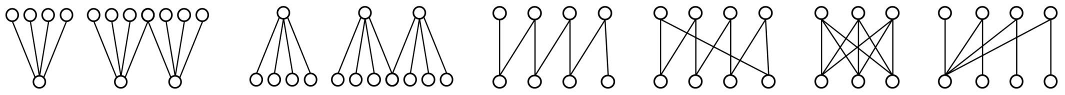

Figure 4. A sequence of eight CBBBs. (All arcs point upward.)

the queue; it enqueues nodes that are newly rendered ELIGIBLE in random order. FIFO is, essentially, the scheduler used by systems such as Condor [4].
Complexity. Each dequeue or enqueue of a single node takes time $O(1)$.

- The LIFO (*last-in, first-out*) scheduler organizes $\mathcal{G}$'s current ELIGIBLE nodes in a stack. It serves a "hungry" client by popping the node at the top of the stack; it pushes nodes that are newly rendered ELIGIBLE onto the stack in random order.
  Complexity. Each push or pop of a single node takes time $O(1)$.
- The STATIC-GREEDY scheduler organizes nodes that are newly rendered ELIGIBLE in a MAX-priority queue whose entries are (partially) ordered by *outdegree*. It serves a "hungry" client by dequeuing the node at the front of the queue. It enqueues nodes that are newly rendered ELIGIBLE in random order; the priority-queue automatically arranges these nodes in decreasing order of outdegree (with ties broken randomly).
  Complexity. Initializing the priority queue takes time $O(n)$; each dequeue of a single node takes time $O(1)$; each enqueue of a single node takes time $O(\log n)$.
- The DYNAMIC-GREEDY scheduler organizes nodes that are newly rendered ELIGIBLE in a structure that is (partially) ordered by nodes' *yields* (with ties broken randomly). The *yield* of an ELIGIBLE node v at time t is the number of non-ELIGIBLE nodes that would be rendered ELIGIBLE if v were executed at this step. DYNAMIC-GREEDY thus makes the same *local* decisions as does an optimal IC-scheduler (when one exists)—but it cannot match the IC-scheduler's tie-breaking foresight. Because the yield of a node changes step by step, the execution of a node u may change the yields of several nodes (specifically, all those that share a child with u), we have implemented DYNAMIC-GREEDY by maintaining the currently ELIGIBLE nodes in a *list of nodes with attached yield-scores*.
  Complexity. Initializing the list takes time $O(n)$. Serving a "hungry" client takes time $O(n)$, using an EXTRACT-MAX operation. After an outdegree-d task v has completed, adding the resulting new ELIGIBLE tasks takes time $O(n)$: (a) We must potentially add all d of v's children to the list of ELIGIBLE nodes; (b) we must update the current yields of all nodes that share a child with v (which could be $O(n)$ in number). This means that DYNAMIC-GREEDY and AO have proportional worst-case computational complexities.

4) The computational platform: The setup described thus far suffices for our AREA-measuring experiment. Our batched-makespan experiment, however, demands a model of the computational platform in which DAGs will be executed. We employ a server-centric model similar to that in [16], the IC-scheduling precursor to this paper. We model the simulated execution of a DAG $\mathcal{G}$ by scheduling heuristic[5] HEUR via a discrete time-ordered queue of "events." Each "event" is represented by the not-yet-executed *residue* of $\mathcal{G}$, together with the current set of ELIGIBLE nodes, organized as mandated by HEUR. The initial residue of $\mathcal{G}$ is $\mathcal{G}$ itself; the initial set of ELIGIBLE nodes comprises $\mathcal{G}$'s sources. The transition from one "event" to its successor proceeds as follows

1) The server polls the available "hungry" clients and allocates one ELIGIBLE task of $\mathcal{G}$ to some of these clients. (Only some "hungry" clients get served because there may not be enough ELIGIBLE nodes to serve them all.) Once allocated, a task is no longer ELIGIBLE.
2) Independently, and asynchronously, the served clients execute their allocated tasks.
3) When a client completes (executing) its allocated task, call it v, the server removes v from the current residue of $\mathcal{G}$ and adds the nodes that are rendered ELIGIBLE by v's completion to the set of current ELIGIBLE nodes, in the manner mandated by HEUR.

(An easy modification of this model would allow for clients that never execute their assigned tasks.)

Our model for the computational platform is completed by specifying two probability distributions, one that describes the arrival pattern of "hungry" clients, and one that describes the completion time of each task.
Client arrivals. At each time-step t of a simulated DAG-execution, we generate a number c_t of "hungry" clients that are seeking tasks at step t. We choose c_t from an exponential distribution.
Serving clients. If e_t nodes of $\mathcal{G}$ are ELIGIBLE at step t, then for each scheduling heuristic HEUR, we select the current highest-priority $\ell = \min(c_t, e_t)$ nodes to assign to the "hungry" clients (using HEUR's priority measure). The server does not know which clients, if any, are more powerful than others, so it treats all clients equally. Possible differences in client power are modeled via the distribution of task execution times.

[5]HEUR $\in$ {AO, FIFO, LIFO, STATIC-GREEDY, DYNAMIC-GREEDY}.

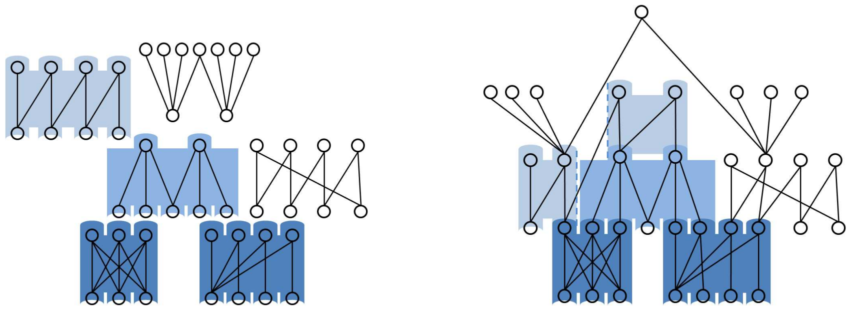

Figure 5. Composing six CBBBs into a LEGO®-DAG: (left) the CBBBs that compose the DAG; (right) the resulting LEGO®-DAG.

Task execution times. The execution time, t, of an allocated task v is chosen randomly from a normal distribution. We model v's execution by inserting into the timeline, at time `current_time`$+t$, the event *task-completion*(v).

When a task completion leaves an empty residue for $\mathcal{G}$, the simulation ends; the `current_time` is stored for subsequent analysis.

B. Experimental Methodology

Client arrivals. We choose the number c_t of "hungry" clients at step t from exponential distributions with rate parameters $\lambda = 1, 1/2, 1/4, 1/8, 1/16, 1/32$. Because the expected value of an exponential distribution with rate parameter λ is just $\mu = 1/\lambda$, our chosen values of λ provide, respectively, 1, 2, 4, 8, 16, and 32 clients per poll on average.

Task execution times. We choose the execution time of an allocated task randomly from a normal distribution with mean 1. We have studied two distributions, one with standard deviation 0.1 and one with standard deviation 0.5. The latter parameter, in particular, allows us to observe the performance of our heuristics on platforms having a rather high level of *heterogeneity*.

DAG *sizes*. Our experiments simulate the execution of DAGs that range in size from 200 nodes to $4,000$ nodes. We thereby observe the performance of our heuristics on DAGs that range from subcomputations to full computations.

Generating random DAG*s having roughly n nodes*. (The specifics of the random processes we use make it hard to specify the number of nodes exactly.)

• *SP*-DAG*s*. We generate a random binary tree $\mathcal{T}$ and randomly designate each internal node of $\mathcal{T}$ either a *series-composition node* or a *parallel-composition node*. We then view $\mathcal{T}$ as the composition tree $\mathcal{T}_\mathcal{G}$ of an n-node SP-DAG.

• *LEGO®*-DAG*s*. While the size of the current DAG is smaller than n, we randomly choose an $(m \leq n)$-node instance $\mathcal{H}$ of one of our six genres of CBBB. Inductively, after processing $i \geq 0$ CBBBs, we have a LEGO®-DAG $\mathcal{G}_i$ that has, say, t targets. We next randomly select some $0 < k \leq t$ sources of $\mathcal{H}$ from a harmonic-like distribution; specifically, the probability of selecting $k = t-h+1$ sources is proportional to $1/h$; we use a harmonic distribution in order to foster connections among CBBBs. We finally merge/identify the k selected sources of $\mathcal{H}$ with a randomly chosen k of $\mathcal{G}_i$'s targets. The resulting LEGO®-DAG is $\mathcal{G}_{i+1}$. The final LEGO®-DAG, $\mathcal{G}$, is achieved when the then-current $\mathcal{G}_i$ has the desired size. Fig. 5 illustrates the initial and final steps of this process.

CBBB sizes. We consider three different families of LEGO®-DAGs, that differ in the way the sizes of the constituent CBBBs (the parameter m) is chosen:

- *Uniform LEGO®*-DAG*s*: the value of m is drawn randomly from the set $[2, 20]$;
- *Exponential LEGO®*-DAG*s*: the value of m is drawn from an exponential distribution with $\lambda = 1/10$ (so that CBBBs have 10 nodes on average);
- *Harmonic LEGO®*-DAG*s*: the value of m is drawn from a harmonic distribution that generates CBBBs having average size 10.

Experimental procedures. For both the makespan-comparison and AREA-comparison experiments, we execute four sets of 45 DAGs each: 5 DAGs of each size $n \approx 200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000$. Each trial involves 100 executions of each DAG. The results on like-sized DAGs are averaged. We then use the means and variances of the schedulers' performances (makespans or AREAs) for our comparisons and analyses.

C. Experimental Results and Discussion

1) Makespan-comparison: The experiment described in this section is intended to evaluate AO-scheduling in a "real" setting, i.e., to do for AO-scheduling what the study in [16] does for optimal IC-scheduling.

We have considered 120 different test settings, each setting characterized by the class of DAGs considered, the scheduling heuristic analyzed, and the rate of arrivals of "hungry" clients. Each test setting is identified by a triple (D, H, μ) where

- D indicates the class of DAGs: $D \in \{$SP-DAGs, Uniform LEGO®-DAGs, Exponential LEGO®-DAGs, Harmonic LEGO®-DAGs$\}$.
- H indicates the scheduling heuristic: $H \in \{$AO, DYNAMIC-GREEDY, STATIC-GREEDY, FIFO, LIFO$\}$.

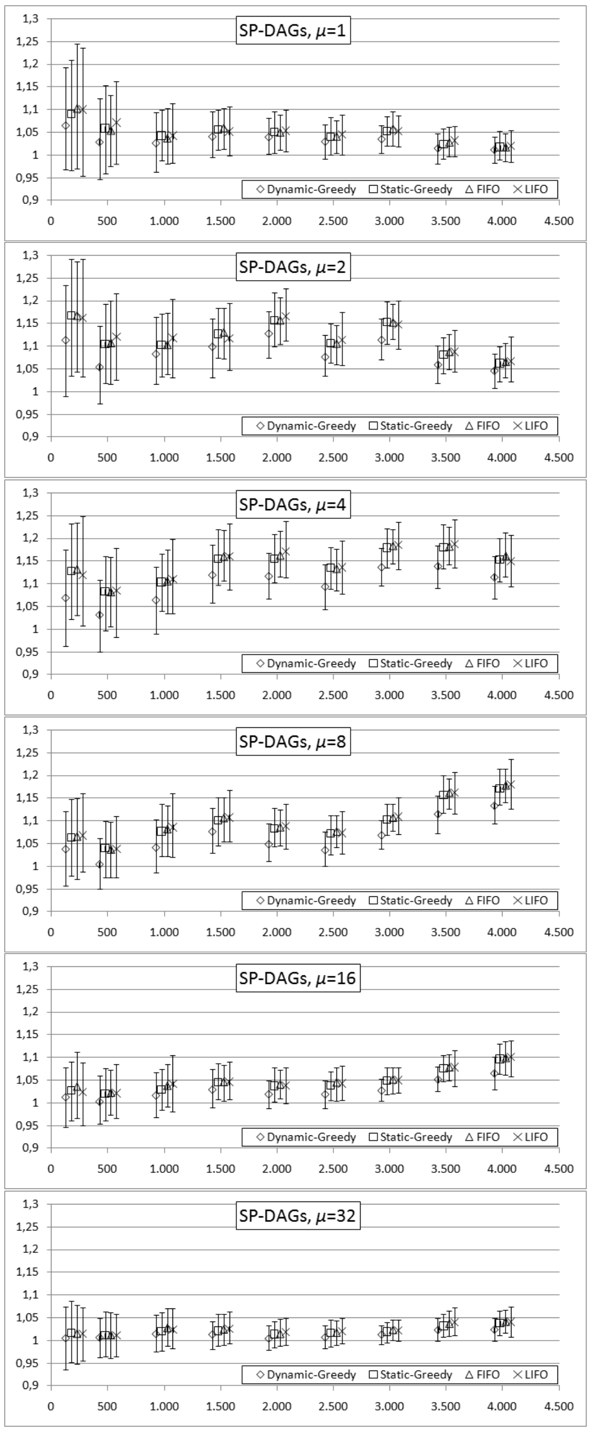

Figure 6. Timing-ratios for *random SP*-DAGs when the average number of “hungry” clients is $\mu = 1, 2, 4, 8, 16, 32$ (top to bottom).

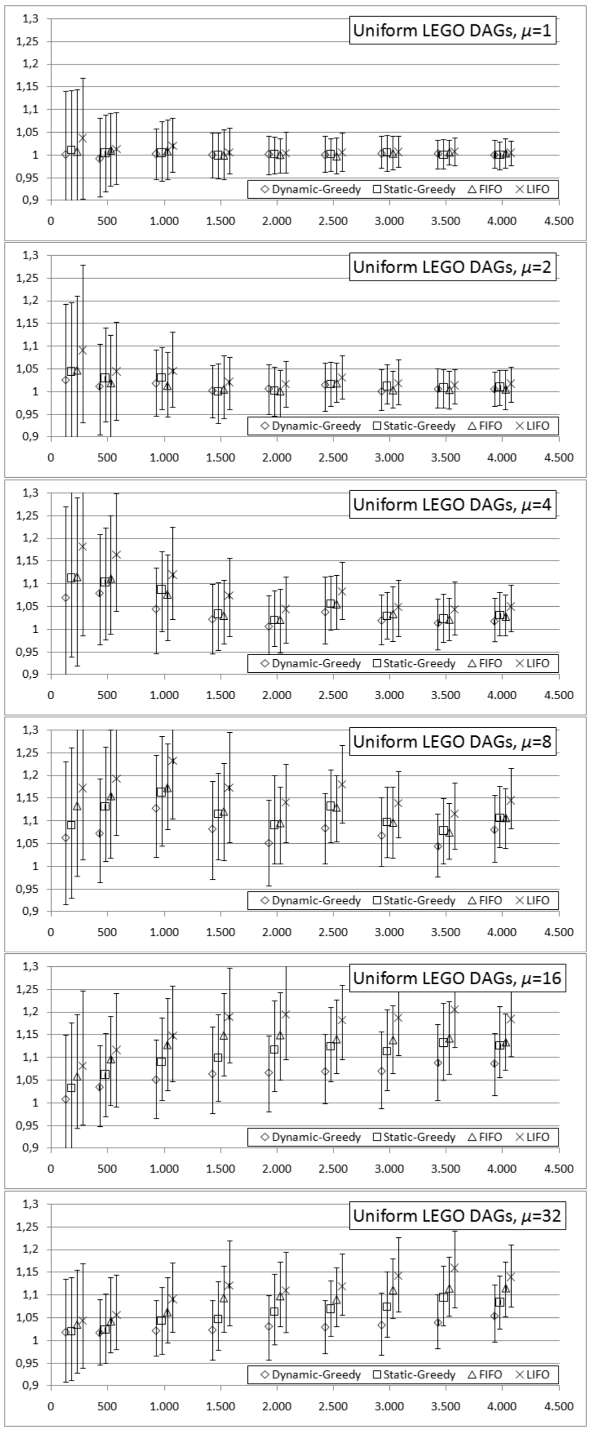

Figure 7. Timing-ratios for *Uniform LEGO*®-DAGs when the average number of “hungry” clients is $\mu = 1, 2, 4, 8, 16, 32$ (top to bottom).

- μ indicates the mean number of “hungry” clients per step: $\mu \in \{1, 2, 4, 8, 16, 32\}$.

For this experiment, the standard deviation of task-execution time is fixed at 0.1. For each test setting, we execute each DAG one hundred times, collecting the final simulation times.

We illustrate the performance of heuristic AO, as compared with its four competitors, via the *timing-ratios* $T(H) \div T(\text{AO})$, where $T(H)$ denotes the simulation time observed using scheduling heuristic $H \in \{$DYNAMIC-GREEDY, STATIC-GREEDY, FIFO, LIFO$\}$. Note that larger values of

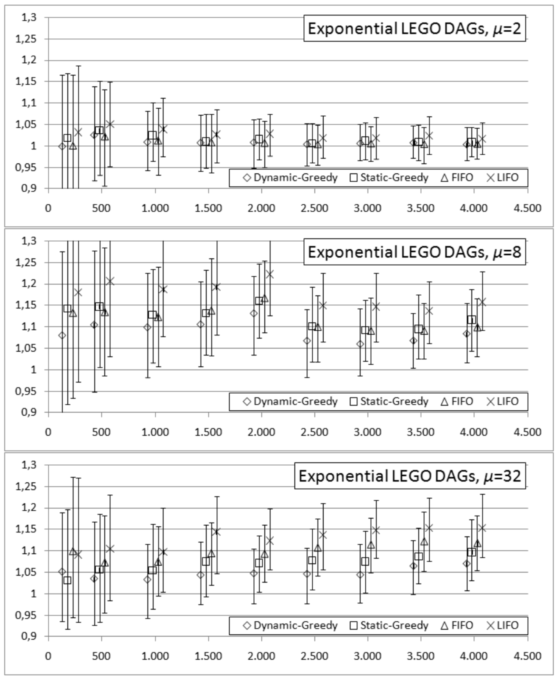

Figure 8. Timing-ratios for *Exponential LEGO*-DAG*s* when the average number of "hungry" clients is $\mu = 2$, 8, 32 (top to bottom).

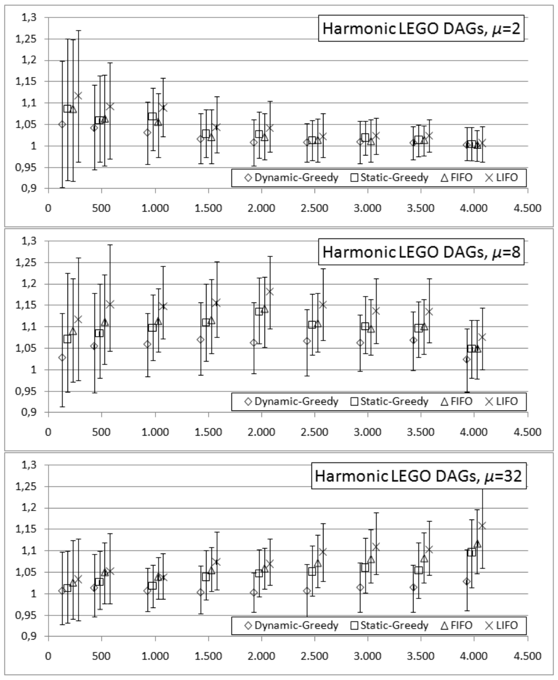

Figure 9. Timing-ratios for *Harmonic LEGO*$^{®}$-DAG*s* when the average number of "hungry" clients is $\mu = 2$, 8, 32 (top to bottom).

the ratio favor heuristic AO. We present both means and 95% confidence intervals in Figs. 6, 7, 8 and 9. To enhance legibility, we present a separate plot for each value of both D and μ. To conserve space, we present the results about random SP-DAGs and *Uniform* LEGO$^{®}$-DAGs with all the values of μ analyzed, whereas the results about *Exponential* and *Harmonic* ones are presented only for $\mu = 2, 4, 8$, solely to indicate that the three families of LEGO$^{®}$-DAGs exhibit very similar behavior; cf. Figs. 7, 8, 9. In each plot, the X-axis indicate DAG-size, while the Y-axis indicates the timing-ratios for the four heuristics that compete with AO.

Our first observation mirrors one from [16]: one observes the benefits of AO-scheduling only for "intermediate" arrival rates μ of "hungry" clients. This is not surprising. When clients arrive very infrequently, i.e., when $\mu \approx 1$, *any* heuristic will require time close to n to execute an n-node DAG; one observes this in the top plots of Figs. 6 and 7. At the other extreme, when clients "flood" the system, there is so much "parallelism" that the only hard limitation for *any* heuristic will be the length of a DAG's inherently sequential "critical path." In both of these extremes, makespan will not depend on the scheduling heuristic.

Between the preceding extremes, though, there is a range of values of μ where the scheduling heuristic has a strong influence on makespan; in our trials, when $1 < \mu \leq 32$, AO always completed executing the DAG in less (simulated) time than its competitors. Importantly, we observed that:

Within a broad range of client arrivals, the makespan of a heuristic, as exposed in Figs. 6, 7, 8 and 9, correlates strongly with the AREAs of the heuristic's schedules, as exposed in Fig. 12. In other words, we observed that schedules with higher AREAs completed executing DAG*s with smaller makespans.*

The *amount* of observed advantage in makespan depended on three factors: the value of μ, the size of the DAG being executed and the family of DAGs. Several cases (e.g., $\mu = 8, 16$) show an improvement in the range of 7–12% for LEGO$^{®}$-DAGs and 10–14% for SP-DAGs. Recall that AO always provides an *AREA-max* schedule for each SP-DAG but not necessarily for each LEGO$^{®}$-DAG.

Comparing the competitors' schedules, we observe that DYNAMIC-GREEDY always outperforms the other competitors by a considerable margin. This is not surprising because DYNAMIC-GREEDY dynamically makes the same local decision as an IC-optimal schedule. In compensation, DYNAMIC-GREEDY is much more demanding computationally than the other heuristics. STATIC-GREEDY and FIFO perform roughly equivalently much of the time, but STATIC-GREEDY sometimes significantly outperform FIFO; e.g., (LEGO$^{®}$-DAGs, STATIC-GREEDY, 16) is much better than (LEGO$^{®}$-DAGs, FIFO, 16). LIFO is always the worst heuristic; for SP-DAGs, though, the three static heuristics: STATIC-GREEDY, FIFO, and LIFO, do not differ substantially.

The impact of client arrival-rates. We have just noted that average client arrival rate μ influences the performance

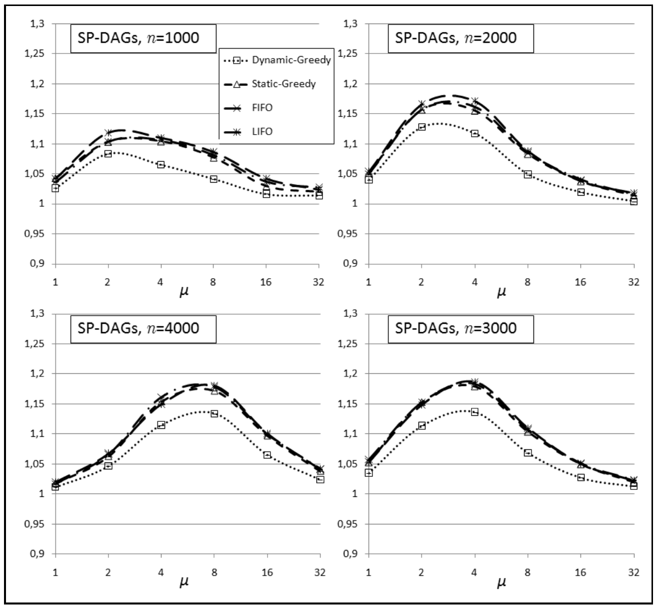

Figure 10. Timing-ratios for random SP-DAGs of different sizes. Clockwise from the top-left: 1000 nodes, 2000 nodes, 3000 nodes, 4000 nodes. The X-axes indicate the average number of "hungry" clients at each poll.

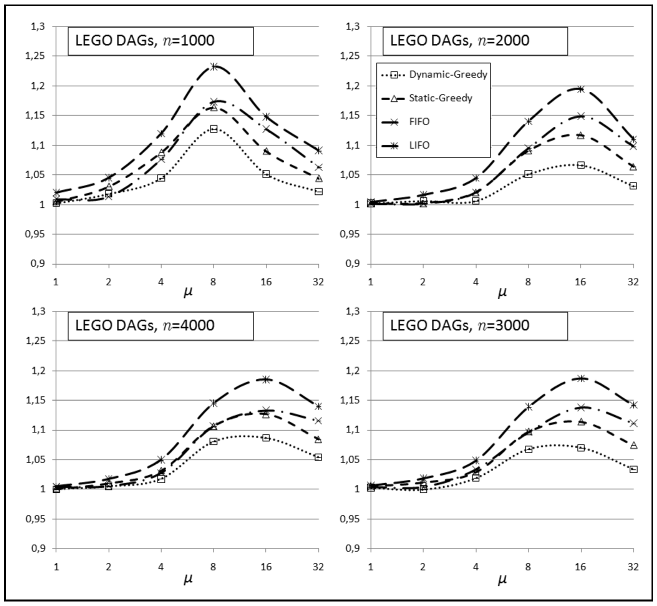

Figure 11. Timing-ratios for random Uniform LEGO®-DAGs of different sizes. Clockwise from the top-left: 1000 nodes, 2000 nodes, 3000 nodes, 4000 nodes. The X-axes indicate the average number of "hungry" clients at each poll.

of AO relative to its competitors. In order to refine this observation, with an eye toward better understanding how μ influences the relative qualities of schedules, we provide, in Figs. 10 and 11, plots that show the performance advantage of AO (in terms of timing-ratios) as a function of μ; the values of μ appear logarithmically along the X-axes of the plots. Both figures present four plots each, depicting the

Table I
THE AVERAGE PARALLELIZABILITY OF SP-DAGS AND LEGO®-DAGS.

	DAG-size (nodes)	DAG-size (arcs)	normalized AREA	Critical Path Length
SP-DAGs	1000	1219	70	150
	2000	2429	75	328
	3000	3666	106	411
	4000	4920	181	445
LEGO-DAGs	1000	2885	76	58
	2000	5644	132	74
	3000	8332	189	92
	4000	11114	255	119

average time-ratios when heuristics execute DAGs having four (approximate): 1000 nodes, 2000 nodes, 3000 nodes and 4000 nodes.

The most notable similarity in the plots is that all are unimodal: for small rates, AO's relative performance improves with increasing μ; this trend continues to a unique peak, after which its relative performance degrades with increasing μ. Moreover, the peak advantage of AO is comparable for DAGs of similar sizes, whether they be LEGO®-DAGs or SP-DAGs. However, there are also notable differences in the plots, particularly between LEGO®-DAGs as a class and SP-DAGs as a class. Specifically we observe the advantage of AO peaking at a higher value of μ for LEGO®-DAGs than for SP-DAGs. Moreover, while the value of μ that maximizes the advantage of AO for SP-DAGs grows roughly linearly with DAG-size (the maximizing values range from 2 for 1000-node DAGs to 8 for 4000-node DAGs), this does not appear to happen with LEGO®-DAGs (the maximizing values there start at 8, for 1000-node DAGs, and then jump to 16 for the other three DAG-sizes).

In an attempt to understand why our two DAG families react differently to the average client arrival rate, we have analyzed certain characteristics of DAGs from these families. Based on our analysis of the data in Table I, we conjecture that *the maximizing value of μ depends on the inherent degree of parallelism provided by the* DAG *being executed.* Notably, the entries in the table show that the DAGs in our two families provide quite different degrees of inherent parallelism. Specifically, LEGO®-DAGs have smaller critical path lengths and a higher normalized AREAs than SP-DAGs. (The observed difference would be even larger if we used optimal AREA-oriented—i.e., AREA-max—schedules for LEGO®-DAGs rather than the often-suboptimal schedules provided by heuristic AO.) Basically, the values of normalized AREA and critical path length show that LEGO®-DAGs are more "parallelizable" than SP-DAGs.

Accommodating heterogeneity by allowing large variance in task execution-times. A major motivation for the development of IC-scheduling (cf. [27])—hence also of AO-scheduling—was the observed *temporal unpredictability* of many modern computing platforms, which precludes the accurate use of classical, critical-path based, DAG-scheduling strategies (cf. [21]). As noted in sources such as [19], [27], we seldom know literally *nothing* quantitative about the computational platform; it is more that our knowledge is very indefinite. A basic tenet of both IC-scheduling and AO-scheduling is that one does not have to deal explicitly with this uncertainty when scheduling a DAG—as long as one enhances the rate of producing ELIGIBLE tasks. We test this tenet in our experiments by allowing great variability in task execution-times, specifically via the variance (or standard deviation) in our model's distribution of these times. How important, though, is the size of the allowed variance? This section seeks guidance on this question.

Our primary model allows 10% deviation in the average task execution-time: a mean time of 1 and a standard deviation of 0.1. How would our results change if we allowed 50% deviation: mean time of 1 and a standard deviation of 0.5? We have repeated all the experiments presented in earlier sections with this new, larger standard deviation. The results are rather surprising. When we increase our model's standard deviation from 0.1 to 0.5—a truly significant change!—the observed relative performance of heuristic AO is almost unchanged! Because the new results are so close to the one we have presented, there would be no value in exhibiting new plots. We have analyzed the relationship between the average makespan obtained with the two standard deviations in task execution-times, 0.1 and 0.5, and have observed that these differences do not exceed 0.05%. Consequently, we can report that the quality of AO-schedules (as generated by heuristic AO) relative to the four competing heuristics is virtually unaffected by both heterogeneity and temporal unpredictability in "task-hungry" platforms.

2) AREA-comparison: The results of our makespan-oriented experiment suggest that AO-scheduling, as implemented by heuristic AO, has a benign impact on computational performance. This inference has led us to wonder:

- How much larger in AREA are the schedules produced by heuristic AO than the schedules produced by its four heuristic competitors?
- How well do the observed differences in the makespan-performance of the four competitors track the differences in the AREAs of schedules produced by these heuristics?

This section is devoted to studying these questions via an experiment that compares the AREAs of schedules produced by heuristic AO to the AREAs of schedules for the same DAGs that are produced by AO's four competitor heuristics. An additional question of interest is motivated by the heuristic AO's dual nature; cf. Section IV-A3.

- By design, heuristic AO operates differently depending on how a DAG $\mathcal{G}$ is presented to it. If $\mathcal{G}$ is presented via a series-parallel decomposition tree, then AO uses the algorithm from [9] to provide an AREA-max schedule for $\mathcal{G}$; else, if $\mathcal{G}$ is presented via some standard presentation of DAGs, say as an adjacency list (see [11]), then AO uses the multi-phase procedure of Section III-B to provide a heuristic approximation to an AREA-max schedule. The question: If we present an SP-DAG $\mathcal{G}$ to heuristic AO in two ways, via a series-parallel decomposition tree and via an adjacency list, how different will be the AREAs of the schedules for $\mathcal{G}$ that AO provides?

We have attempted to answer these question via an experiment that considers 20 different test settings. each characterized by the class of the DAGs considered and the scheduling heuristic analyzed. For each test setting we execute each DAG 100 times, collecting the schedule's AREA value. Fig. 12 presents the mean recorded AREA values, as well as the ranges $[\min, \max]$. The figure presents one plot for different-size instances of each of the families of DAGs indicated in the figure caption; the sizes of the DAG-instances appear along the X-axes.

Results from Fig. 12. Of course, for SP-DAGs, the schedules provided by the AO scheduler, being AREA-max always have the largest AREAs. As hoped, for general DAGs, the AREA-superiority of schedules provided by the AO heuristic persists. This second observation suggests that the procedure in Section III-B that defines the AO heuristic works very well in terms of AO-scheduling. This suggestion is reinforced by the fact that difference between the AREAs of scheduled provided by the AO heuristic and those provided by the competitor heuristics grows more than linearly with the size of the DAG being scheduled.

Considering all of experimental results, as exposed in Figs. 6, 7, 10, and 12, we provide three observations that support our hypothesis that *there is a strong positive relation between the AREA of a schedule and its makespan* (at least as simulated by our makespan experiment).

- The schedules provided by all five heuristics—AO and its competitors—*have the same relative ranking in the makespan and AREA experiments;* i.e., statistically, AO outperforms DYNAMIC-GREEDY, which outperforms STATIC-GREEDY, which outperforms FIFO, which outperforms LIFO.
- When used on SP-DAGs, *the schedules provided by the* STATIC-GREEDY, FIFO, *and* LIFO *heuristics have roughly the same AREA, as well as roughly the same makespan.*
- The ratio between the (*optimal*) AREAs of schedules provided by the AO heuristic and the AREAs of schedules provided by the four competitor heuristics is roughly 4 for SP-DAGs and only roughly 2 for LEGO®-DAGs. *This correlates positively with the relative improvements in makespan for the same families of* DAGs.

In the interest of full disclosure, we do not yet know if the observed differences between results for SP-DAGs

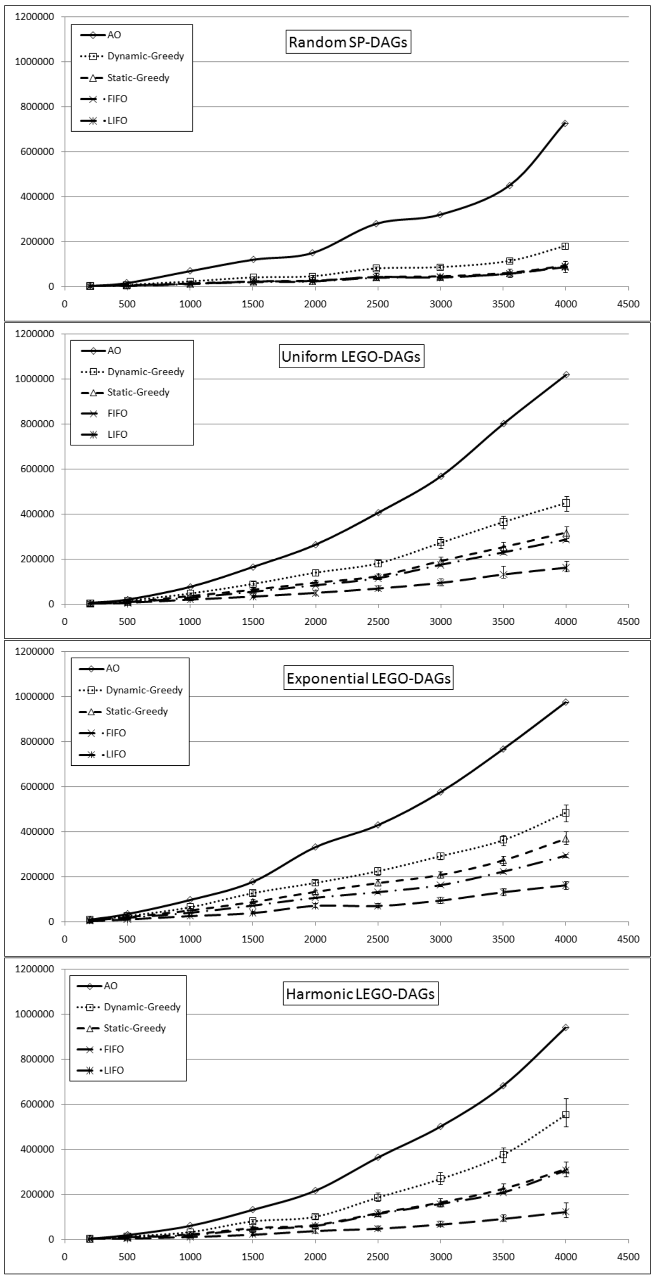

Figure 12. AREA comparison. From top to bottom: Random SP-DAGs, Uniform LEGO®-DAGs, Exponential LEGO®-DAGs, Harmonic LEGO®-DAGs.

and for LEGO®-DAGs are *inherent*, due to the different characteristics of such DAGs (cf. Table I), or algorithmic, due to a possible loss of quality introduced by the heuristics of Section III-B.

V. Conclusion

Building on the novel *AREA-oriented* (*AO*) scheduling paradigm of [8], we have assessed the quality of AO-schedules for a variety of artificially generated DAGs whose structures are reminiscent of those encountered in real scientific computations. The hope is that the rate at which AO-schedules produce DAG-nodes that are eligible for allocation to clients will make such schedules computationally advantageous for modern "task-hungry" computational platforms, such as Internet-based platforms, aggressively multi-core platforms, and exascale platforms.

Our assessment pitted our new efficient heuristic, AO, for producing AO-schedules against four common scheduling heuristics that represent different points in the sophistication-complexity space of schedulers. We have shown via simulation experiments that

- The schedules produced by AO have *AREAs that are closer to optimality* than are the schedules produced by the four competing heuristics.
- The schedules produced by AO have *lower makespans* than do the four competing heuristics, based on a probabilistic model of the computational platform and the DAG-executing process.

Importantly, our experiments suggest that there is a strong positive relationship between the AREA of a DAG-schedule and the schedule's performance, as measured by its makespan.

We view the new scheduling heuristic, AO, *which operates within time quadratic in the size of the* DAG *being scheduled*, as an important advance because:

- The problem of finding truly AREA-maximizing schedules is likely to be computationally intractable [8].
- AO represents the first efficient scheduling mechanism that provably enhances the rate of producing allocation-eligible nodes for *every* computation-DAG.

Finally our experiments have a high degree of *robustness*. The demonstrated computational benefits of AO-scheduling persist even when the "task-hungry" platforms have a high degree of heterogeneity and/or a high degree of temporal unpredictability. (We model both of these scheduling challenges by allowing a large variance in task execution-times within our probabilistic model.)

Where we are going. Our demonstration of the computational benefits of AO-scheduling reinforces the importance of the two algorithmic questions raised in Section III-B.

- Does there exist an algorithm for crafting AREA-max schedules for SP-DAGs that is more efficient than the quadratic-time algorithm of Lemma 3.2?
- Does there exist an algorithm for SP-izing arbitrary DAGs whose use would improve the AREAs and makespans of schedules provided by our AO-scheduling heuristic AO?

Additionally, the "success" of our experiments suggests the desirability of assessing the value of AO-scheduling via experiments with real computations rather than simulated artificial ones. We hope to follow this path in the not-distant future.

Acknowledgement: The research of A. Rosenberg was supported in part by NSF Grant CNS-0905399. The authors thank A. Gonzáles-Escribano and his team for providing access to their DAG-SP-ization code.

References

[1] D. Adolphson and T.C. Hu (1973): Optimal linear ordering. *SIAM J. Appl. Math. 25*, 403–423.

[2] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou (1995): Cilk: An efficient multithreaded runtime system. *5th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP'95).*

[3] R.D. Blumofe and C.E. Leiserson (1998): Space-efficient scheduling of multithreaded computations. *SIAM J. Comput. 27*, 202–229.

[4] Condor Project, Univ. of Wisconsin, http://www.cs.wisc.edu/condor.

[5] G. Cordasco, G. Malewicz, A.L. Rosenberg (2007): Advances in IC-scheduling theory: scheduling expansive and reductive dags and scheduling dags via duality. *IEEE Trans. Parallel and Distributed Systems 18*, 1607–1617.

[6] G. Cordasco, G. Malewicz, A.L. Rosenberg (2007): Applying IC-scheduling theory to some familiar computations. *Wkshp. on Large-Scale, Volatile Desktop Grids (PCGrid'07).*

[7] G. Cordasco, G. Malewicz, A.L. Rosenberg (2010): Extending IC-scheduling via the Sweep algorithm. *J. Parallel and Distributed Computing 70*, 201–211.

[8] G. Cordasco and A.L. Rosenberg (2009): On scheduling DAGs to maximize area. *23rd IEEE Int'l Parallel and Distr. Processing Symp. (IPDPS'09).*

[9] G. Cordasco and A.L. Rosenberg (2010): Area-maximizing schedules for series-parallel DAGs. *16th Int'l Conf. on Parallel Computing (EURO-PAR'10)*, Part II. In *Lecture Notes in Computer Science 6272*, Springer, Berlin, 380–392.

[10] G. Cordasco, A.L. Rosenberg, M. Sims (2008): On clustering tasks in IC-optimal DAGs. *37th Int'l Conf. on Parallel Processing (ICPP'08).*

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein (1999): *Introduction to Algorithms* (2nd ed.). MIT Press, Cambridge, MA.

[12] J. Dongarra et al. (2010): International Exascale Software Project Roadmap. Tech. Rpt. UT-CS-10-652, Univ. Tennessee.

[13] A. González-Escribano, A. van Gemund, V. Cardeñoso-Payo (2002): Mapping unstructured applications into nested parallelism. *High Performance Computing for Computational Science (VECPAR '02).*

[14] A. González-Escribano, A. van Gemund and V. Cardeñoso-Payo (2009): Performance implications of synchronization structure in parallel programming. *Parallel Computing 35 (8-9)*, 455–474.

[15] I. Foster and C. Kesselman [eds.] (2004): *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition)*. Morgan-Kaufmann, San Francisco.

[16] R. Hall, A.L. Rosenberg, A. Venkataramani (2007): A comparison of DAG-scheduling strategies for Internet-based computing. *21st IEEE Int'l Parallel and Distr. Processing Symp. (IPDPS'07)*

[17] L. He, Z. Han, H. Jin, L. Pan, S. Li (2000): DAG-based parallel real time task scheduling algorithm on a cluster. *Int'l Conf. on Parallel and Distr. Processing Techniques and Applications*, 437–443.

[18] S. Jayasena and S. Ganesh (2003): Conversion of NSP DAGs to SP DAGs. MIT Course Notes 6.895.

[19] D. Kondo, H. Casanova, E. Wing, F. Berman (2002): Models and scheduling mechanisms for global computing applications. *Int'l Parallel and Distr. Processing Symp. (IPDPS'02)*

[20] E. Korpela, D. Werthimer, D. Anderson, J. Cobb and M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Science and Engineering* (P.F. Dubois, Ed.) IEEE Computer Soc. Press.

[21] Y.-K. Kwok and I. Ahmad (1999): Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys 31*, 406–471.

[22] G. Malewicz, I. Foster, A.L. Rosenberg and M. Wilde (2007): A tool for prioritizing DAGMan jobs and its evaluation. *J. Grid Computing 5*, 197–212.

[23] G. Malewicz and A.L. Rosenberg (2005): On batch-scheduling dags for Internet-based computing. *11th Int'l Conf. on Parallel Computing (EURO-PAR'05).* In *Lecture Notes in Computer Science 3648*, Springer, Berlin, 262–271.

[24] G. Malewicz, A.L. Rosenberg and M. Yurkewych (2006): Toward a theory for scheduling dags in Internet-based computing. *IEEE Trans. Comput. 55*, 757–768.

[25] M. Mitchell (2004): Creating minimal vertex series parallel graphs from directed acyclic graphs. *2004 Australasian Symp. on Information Visualisation 35*, 133–139.

[26] C.H. Papadimitriou and M. Yannakakis (1991): Optimization, approximation, and complexity classes. *J. Computer and System Scis. 43*, 425–440.

[27] A.L. Rosenberg (2004): On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput. 53*, 1176–1186.

[28] A.L. Rosenberg and M. Yurkewych (2005): Guidelines for scheduling some common computation-dags for Internet-based computing. *IEEE Trans. Comput. 54*, 428–438.

[29] S. Tomov, J. Dongarra, M. Baboulin (2010): Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing 36 (5-6)*, 232–240,

[30] J. Valdes, R.E. Tarjan and E.L. Lawler (1982): The recognition of series-parallel digraphs. *SIAM J. Comput. 11*, 289–313.