Working paper

Dynamic Data Structures for Taskgraph Scheduling Policies with Applications in OpenCL Accelerators

Jakub Mareček · Andrew J. Parkes · Edmund K. Burke · Robert Elliot · Hedley Francis · Anton Lokhmotov

Abstract OpenCL is an emerging open framework for parallel programming in heterogenous systems. Devices compliant with OpenCL need to schedule the execution of submitted jobs with no (or only very imprecise) estimates of execution times, but respecting dependencies among them, which are given in the form of directed acyclic graph. This problem is known as stochastic taskgraph scheduling, stochastic scheduling with precedencies, or stochastic scheduling with data dependencies.

We study the complexity of implementing static out-of-order policies for taksgraph scheduling, which approach optimality in the long run, under certain assumptions. We present a simple data structure allowing for the "what next" query of such scheduling policies to be answered in time $O(1)$, while vertices can be added in time $O(1)$.

1 Introduction

In stochastic taskgraph scheduling, policies which run jobs with the largest sum of expected processing times along a path in the dependency graph, out of those available for processing, first perform particularly well. Papadimitriou and Tsitsiklis [20] have shown that, under certain restrictions, they are asymptotically optimal. This is not particularly surprising, as they correspond to the well-known critical path heuristics [12] in the off-line case. An important question remains, though: how efficiently can one implement such policies?

The implementation of "longest-path" policies requires a dynamic data structure for maintaining a criterion related to paths in a vertex-weighted directed acyclic graph (DAG), subject to the insertion of a sink with adjacent edges and deletion of a source that maximises the criterion, together with adjacent edges. Let us assume edge $u \to v$ represents the requirement that job u finishes before job v can start. This criterion

Jakub Mareček is the contact author. E-mail: jakub@marecek.cz.

Jakub Mareček · Andrew J. Parkes · Edmund K. Burke
School of Computer Science, The University of Nottingham, Nottingham NG8 1BB, UK
E-mail: { jxm, ajp, ekb }@cs.nott.ac.uk

Robert Elliot · Hedley Francis · Anton Lokhmotov
ARM Ltd., 110 Fulbourn Road, Cambridge CB1 9NJ, UK
E-mail: { Robert.Elliot, Hedley.Francis, Anton.Lokhmotov } @arm.com

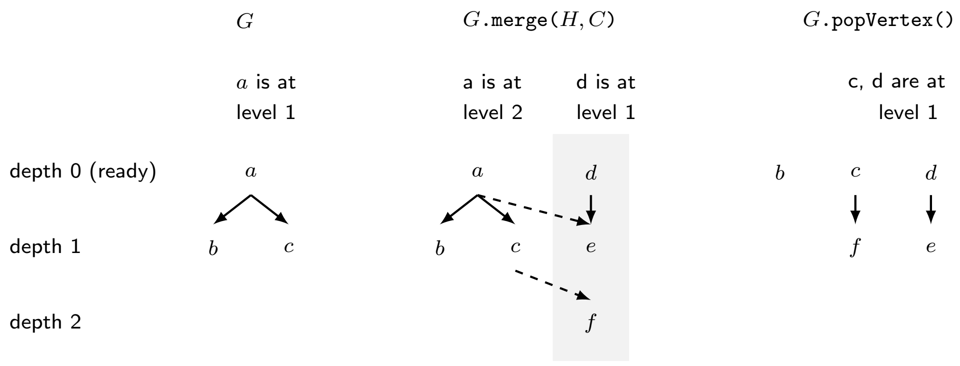

Fig. 1 An illustration of the effects of the required operations: Directed graph $G = (\{a, b, c\}, \{a \to b, a \to c\})$ is merged with directed graph $H = (\{d, e, f\}, \{d \to e\})$ (shaded) with cross edges $C = \{a \to e, c \to f\}$ (dashed). Vertex a has the highest level of 2. When a is removed, three connected components remain, with both vertices c and d at level 1.

known as "level" is typically the length of the longest out-going path, i.e. the longest path in the subgraph reachable from the given source, possibly weighted by weights on the vertices. This should be contrasted with the "depth", which is the length of the longest in-coming path in any vertex. (See Figure 1 for an example.) The required operations are:

– `popVertex`: delete the vertex with no in-going edges that maximises the criterion.
– `merge`(H, C): add (possibly smaller) DAG H to the current (possibly larger) DAG G. There will be no edges from H to G, but there are $|C|$ "cross" edges from G to H passed separately as C.

We also study the following special cases of the two operations:

– "What next" query, picking the (source) vertex with no in-going edges, which maximises one of the criteria mentioned above.
– Deletion of a (source) vertex with no in-going edges, together with all adjacent out-going edges.
– Insertion of a (sink) vertex and a number of in-going edges.

As far as we know, no dynamic data structure supporting even the special cases of the two operations efficiently has been studied previously. There are, however, dynamic data structures for maintaining topological order in DAGs [10] subject to edge insertion and deletion, dynamic data structures for maintaining the length of the shortest or longest in-going path in a DAG [11] subject to edge insertion and deletion, and numerous data structures for maintaining related information in undirected trees subject to a wide range of operations [2, 8, 26]. In Section 3, we present the data structure, together with an analysis in the input and output model of Ramaling and Reps [24], summarised in Table 1. First, however, we introduce the motivating problem in the Section 2, with the related work on taskgraph scheduling policies summarised in Subsection 2.4.

Table 1 Data structures for out-of-order static scheduling policies and the corresponding upper bounds on the complexity of key operations: worst case analysis of a trivial use of linked lists compared to the input and output model analysis [24] of the proposed data structure. Note n is the number of vertices, $|\delta|$ is the number of vertices on the affected longest paths, and $||\delta||$ is the number of vertices in the neighbourhoods of the vertices along the affected longest paths, in all three cases in the resulting graph, and $Q(n)$ is the complexity of insertion and extraction of an element from a priority queue with n elements, which is $\sqrt{\log \log n}$ (amortised) for melding priority queues [17].

Operation	Linked lists	This paper
`merge`(H, C)	$O(\|C\|n)$	$O(\|C\| \log \|C\| + \|\delta\|Q(\|\delta\|) + \|\|\delta\|\|)$
`popVertex`	$O(n^2)$	$O(\|\delta\|Q(\|\delta\|) + \|\|\delta\|\|)$
"What next" query	$O(n^2)$	$O(1)$
Sink insertion	$O(1)$	$O(1)$
Edge-to-sink insertion	$O(n)$	$O(\|\delta\|Q(\|\delta\|) + \|\|\delta\|\|)$
Source deletion	$O(n)$	$O(\|\delta\|Q(\|\delta\|) + \|\|\delta\|\|)$

2 The Motivating Problem

OpenCL is being developed by AMD, Apple, ARM, Intel, Motorola, Nokia, NVidia, Qualcomm, Samsung, Sony, Sun, Texas Instruments, and a number of others, as an open, royalty-free standard for parallel programming in heterogenous systems. The OpenCL specification envisions the bulk of demanding computations being handled by an OpenCL accelerator, rather than the central processing unit (CPU). The OpenCL accelerator maintains a "system queue", which describes what jobs are to be executed and what are the acyclic dependencies among them. This work load comes from a number application, but only indirectly. Each application can have multiple pieces of OpenCL-accelerated code, or "kernels", and a number of "local queues" each. Each "kernel" can be executed multiple times and we denote each execution as a "job". (This is important, as one may expect multiple execution of the same kernel to have similar properties, including run-time.) Jobs are first submitted to the "local queue", where the application can specify further acyclic dependencies among the jobs in the queue, as well as dependencies on the jobs in the system queue. At the point when a "local queue" is "flushed", the scheduler takes over the control. We assume that at most one "kernel" can be run on the OpenCL accelerator at any point in time, which is realistic in embedded applications. If the scheduling was sequential and in-order, jobs in a local queue could be sorted topologically after its flushing, and dependencies between jobs could be disregarded in the "system queue". When the scheduler requires parallel or out-of-order execution, however, acyclic dependencies between jobs need to be stored and checked, before a job is run, or maintained otherwise. Considering that the accelerators are massively parallel and out-of-order execution may be required to accommodate latency constraints, the latter is the case. Notice that, unlike in games consoles where the application and hardware specification is known [3], one cannot easily pre-compute a schedule of an unknown workload. Note that thousands or millions of jobs may well need to be processed per second, and the overhead due to the scheduler needs to be kept to a minimum, especially in battery-powered applications. One, hence, needs a fast implementation of a policy for stochastic task-graph scheduling.

2.1 The Problem

In stochastic task-graph scheduling, we assume that there exists an unknown vertex-weighted directed acyclic graph, "taskgraph", with countably many nodes. Vertices of the taskgraph correspond to jobs and there is an edge between vertices u, v, if and only if job v must be run only after completion of job u. The vertex weights in the taskgraph correspond to run-times of the jobs. Jobs (vertices) are revealed in batches, together with some estimates of their run-times. Jobs, which do not have any dependencies, are called "available". There are also m identical machines, each of which can process any of the available jobs.

A policy is a rule for deciding, which of of one of the available jobs is to be run on a machine. In particular, we are interested in the families of:

- non-delay policies, where a job is executed, whenever there are jobs available and there is capacity to process at least one of them
- non-anticipative policies, where no assumptions are made about jobs arriving in the future
- non-preemptive policies, where a job is run until completion, whenever it is run.
- preemptive policies with preemption cost c_p, where a job can be stopped at any point and resumed after a preemption routine, whose runtime is c_p.

In practice, there are considerable preemption costs in preemptive policies. This preemption routine needs to save a great number of registers and may increase the number of cache misses, which may be comparable to running the average job, and hence make preemptive policies with realistic preemption costs very close to non-preemptive policies.

In the long-run horizon, we study the throughput and (discounted) weighted makespan of policies. More formally, we focus on:

- the weighted throughput of the system, given by policy π:

$$J(\pi) = \limsup_{t \to \infty} \frac{1}{t} \sum_{q \in Q} w(q) \mathbb{E}[a_q^\pi(t)]$$

- the α-discounted weighted makespan of the system, given by policy π:

$$K(\pi) = \sum_{q \in Q} \sum_{i=1}^{\infty} w(q) \mathbb{E}[e^{-\alpha C_i(q)}]$$

where a_q is the number of jobs from queue q completed by time t, and $C_i(q)$ is the completion time of ith job in queue q, both of which are well-defined random variables, and $0 < \alpha \leq 1$ is the discount rate.

Given an objective function f, input σ, and the optimum of f on σ, $OPT_f(\sigma)$, the asymptotic approximation (performance, competitive) ratio of policy π of is:

$$R_f^\infty(\pi) = \limsup_{n \to \infty} \left\{ \frac{\pi_f(\sigma)}{OPT_f(\sigma)} \mid OPT_f(\sigma) = n \right\}.$$

Within a family of policies P, policies with approximation ratio:

$$R_f^\infty = \inf_{\pi \in P} R_f^\infty(\pi).$$

are asymptotically optimal.

2.2 The Complexity

The problem of scheduling the stochastic OpenCL Task System and its precisely revealed deterministic snapshot, called the Precisely Revealed OpenCL Task System, has been formalised and studied in another paper by the authors [16]. When one formalises the problems, it is easy to see that the problem of deciding the non-preemptive schedulability of a Precisely Revealed OpenCL Task System within a finite time horizon T is $\mathcal{NP}$-Hard. Indeed, non-preemptive schedulability of jobs with precedencies on two or more machines within a finite time horizon has been on Karp's original list of $\mathcal{NP}$-Complete problems [9]. Perhaps more interestingly, deciding if there is a non-preemptive policy resulting in priority-weighted throughput larger than k for a (Forgetful) OpenCL Task System in an OpenCL Acclerator is $\mathcal{EXPTIME}$-Hard, if there is a fixed assignment of queues to cores. For closed queuing systems (Forgetful OpenCL Task System), one can use the reduction to NetworkOfQueues of Papadimitriou and Tsitsiklis [19,21]. For open queuing systems (OpenCL Task System), one needs to prove a similar result for a variant of NetworkOfQueues. The problem is hence hard, independently of any unproven conjectures, such as $P \neq NP$. We can show, however, that there are asymptotically optimal policies, under certain assumptions.

2.3 The Stability

Let us us consider the limits of stability of a closely related system. We define the system to be in a "transient state" ("choked"), if the number of jobs waiting in any queue goes to the large limit in the long run. We define the system to be "stable" if for all queues, there is a finite bound on the expected interval between two times when the number of work-groups waiting to be processed is zero. It is clear that the stability depends on the arrival rates of jobs, their processing times, and dependencies between work-groups.

In particular, we are interested in the effects of dependencies between jobs. We assume that job j depends on any enqueued jobs with probability p, independently of any other dependencies. λ^* is the best possible arrival rate, that is the interval between the arrival of two jobs. Then:

Theorem 1 (Hajek [27]) $\lim_{p \to 0} \lambda^* p = e^{-1}$

Let us now study the $n - m$ work-groups waiting to be processed, out of n work-groups seen so far. $G[m, n]$ is the subgraph of the taskgraph induced by vertices $\{m, m+1, \ldots, n\}$. We denote d_{mn} the length of longest path in $G[m, n]$. Further, $\beta_n = \mathbb{E}[d_{1n}]/n$. Then:

Theorem 2 (Tsitsiklis et al. [27]) *The limit* $\lim_{n \to \infty}(d_{mn}/n)$ *exists almost surely. Let us use* β^* *to denote the limit for any* m *where it exists. If* $\lambda < 1/\beta^*$*, the system is stable.*

The results below are not conditional on the system being in a stable state, but draw a distinct inspiration from the dependency of the performance on the length of the longest path in the data dependency graph.

2.4 A Policy and Conditions of its Asymptotic Optimality

Let us now consider the non-delay non-preemptive policy scheduling the available job that corresponds to the root in the taskgraph maximising the numbers of jobs along the longest path in the subgraphs reachable from the root, or "level", whenever there are available jobs and a core becomes available. Let us assume that:

- there are jobs with independent identically distributed processing times, drawn from a common binomial or Poisson distribution
- data dependency graph G is a forest of in-trees, and the non-directed counterpart of the taskgraph is hence acyclic.

When we denote these assumptions by an asteriks (*), it has been shown:

Theorem 3 (Papadimitriou and Tsitsiklis [20]) *Processing the job with the highest level ("largest sum of expected processing times along a path in the dependency graph") first, as soon as any machine is available, is asymptotically optimal with respect to weighted throughput, under certain conditions (*), among non-anticipative non-delay non-preemptive policies and non-anticipative non-delay preemptive policies with zero cost of preemption.*

This means that the makespan achieved by the "largest sum of expected processing times along a path in the dependency graph"-first policy is no larger than the optimal makespan by a factor that goes to one in the large limit of the number of jobs. The proof is based on an exchange argument, due initially to Papadimitriou and Tsitsiklis [20] and extended by Liu and Sanlaville [14,15,15]. Similar techniques had appeared in the proofs of Pinedo and Weiss [22] and Chandy and Reynolds and Bruno [4], which were all restricted to two identical processors. Notice that if there were multiple processors running different jobs and arbitrary restrictions on jobs running on certain servers, the policy would no longer be even asymptotically optimal [25].

A number of important questions remain, however: What is the run-time of the corresponding query and is it offset by the improvements over a trivial (first-come first-served) scheduler? The problem of finding the longest path in either general graphs or digraphs is clearly $\mathcal{NP}$-Hard, as the HPP [9] is a special case. The corresponding decision problem in directed acyclic graphs is, however, equivalent to finding the shortest path in undirected graph, and hence in $\mathcal{NL} \subseteq \mathcal{P}$. In order to make the answer more precise, we need to present the corresponding data structures.

3 The Fully-Dynamic Data Structure

In Algorithms 1–7, we present a dynamic data structure for maintaining both the length of the longest in-coming path ("depth) in each node of a directed acyclic graph and and the length of the longest out-going path ("level") in each vertex with no in-coming edges. Unlike in Section 1, where the distinction between the storage of the graph and additional details has been obscured, this section makes the distinction clear: The graph is stored as an adjacency list denoted G, while the additional details are pointers to unweighted level in array G_{UL}, the weighted level G_{WL}, and a priority queue R keeping track of the ready jobs, with respect to the criterion chosen.

The approach for maintaining the data is based on the "contraption under gravity" view developed by Dijkstra in the 1960s [7,5,11]: First, we traverse the out-going

Algorithm 1 `insertEdge`$(G,\, G_{UL},\, G_{WL},\, R,\, u \to v)$

Input: Digraph G, weighted and unweighted auxiliary structures G_{UL}, G_{WL}, priority queue R, edge $u \to v$ to be added
Effect: Updated G, G_{UL}, G_{WL}, R
$G = G \cup \{u \to v\}$
if has$(R,\, v)$ **then**
  remove$(R,\, v)$
end if
$U =$ insertEdgeUpdateDownstream$(G_{UL},\, u,\, v)$
$U = U\cup$ insertEdgeUpdateUpstream$(G_{UL},\, u,\, v)$
$U = U\cup$ insertEdgeUpdateDownstream$(G_{WL},\, u,\, v)$
$U = U\cup$ insertEdgeUpdateUpstream$(G_{WL},\, u,\, v)$
while $U \neq \emptyset$ **do**
  $u =$top(U) // u for updated
  update(R, u)
end while

Algorithm 2 `merge`$(G,\, G_{UL},\, G_{WL},\, R_1,\, H,\, H_{UL},\, H_{WL},\, R_2,\, C)$

Input: Digraphs G, H, weighted and unweighted auxiliary structures $G_{UL}, G_{WL}, H_{UL}, H_{WL}$, priority queues R_1, R_2 related to graphs G, H, respectively, set C of edges $g \to h$ from vertex-set of G to vertex-set of H
Effect: Structures G, G_{UL}, G_{WL}, R_1 are updated
$G = G \cup H$, $G_{UL} = G_{UL} \cup H_{UL}$, $G_{WL} = G_{WL} \cup H_{WL}$
$R_1 = R_1 \cup (R_2 \setminus \{h \mid (g \to h) \in C\})$
$Q =$ queue$\langle$ SumWeightedLongestFirst $\rangle(C)$
while $Q \neq \emptyset$ **do**
  $(g \to h) =$dequeue(Q)
  `insertEdge`$(G,\, G_{UL},\, G_{WL},\, R,\, g \to h)$
end while

subgraph ("down") from the affected vertex along a topological order of vertices, in order to update the depth. Next, we traverse the in-coming subgraph in the reverse direction ("up") along a topological order of vertices, updating the level as long as it is necessary. This can be visualised as a "contraption" of strings and knots held at the knot corresponding to the affected vertex, updating the depth, holding the bottom most knot, and updating the level. The non-trivial part is the maintenance of the topological order of vertices [10], so as to guarantee that no edge is visited twice.

Katriel et al. [11] have analysed the use of a priority queue with the priority being the depth of the vertex in order to obtain the topological order, and applications in maintaining the longest in-coming paths (depth) in directed acyclic graphs subject to edge insertion and deletion. The analysis has been subsequently improved [13,1]. In Algorithms 1–7, we adapt the approach of Katriel et al. [11] to the maintenance of both the length of the longest in-coming path (depth) and the length of the longest out-going path (level) in each node of a vertex-weighted directed acyclic graphs under the operations required in the out-of-order static scheduling policies. Alternatively, one could maintain only the lengths of the longest out-going path (level). This would allow for faster run-times of arbitrary edge insertion, which we do not require, but which would slow down the run-time of `merge` by the all-pairs shortest paths computation.

Let us now analyse the algorithms in the input and output model of Ramaling and Reps [24], using melding priority queues [17]. In order to implement the merge operation (Algorihtm 2), one needs to implement the traversal up and down the directed graph.

Algorithm 3 `popVertex`(G, G_{UL}, G_{WL}, R)

Input: Digraph G, weighted and unweighted auxiliary structures G_{UL}, G_{WL}, priority queue R
Effect: Structures G, G_{UL}, G_{WL}, and R are updated
$v = \mathrm{top}(R)$
$U = \mathrm{popVertexUpdateDownstream}(G_{UL},\, v)$
$U = U \cup$ popVertexUpdateDownstream$(G_{WL},\, v)$
$S = \mathrm{succ}(G_{UL},\, v)$
while $S \neq \emptyset$ **do**
    $s = \mathrm{top}(S)$ // s for successor
    $R.\mathrm{push}(s)$
end while
while $U \neq \emptyset$ **do**
    $u = \mathrm{top}(U)$ // u for updated
    $R.\mathrm{update}(u)$
end while
$G = G \setminus \{u \to v\}$

This is rather straightforward, with pseudocode exhibited in Appendix A. When one denotes $|\delta|$ the number of vertices whose depth or level has changed and $||\delta||$ is the cardinality of the union of neighbourhoods of vertices whose depth or level has changed, one can see:

Claim `insertEdgeUpdateDownstream` runs in time $O(||\delta|| + |\delta|Q(|\delta|))$, `insertEdgeUpdateUpstream` runs in time $O(||\delta|| + |\delta|Q(|\delta|))$, and `insertEdge` runs in time $O(||\delta|| + |\delta|Q(|\delta|))$, where $Q(n)$ is the complexity of insertion and extraction of an element in a priority queue with n elements.

Proof sketch: The run-time complexity of `insertEdge` is dominated by the complexity of `insertEdgeUpdateDownstream`. There, each vertex is inserted and extracted from the queue at most once, and all $||\delta||$ neighbouring vertices need to be checked.

Consequently:

Lemma 1 `merge`*(A,B) runs in time* $O(c \log c + |\delta|Q(|\delta|) + ||\delta||)$*, where c is the number of "cross" edges from B to A, δ corresponds to changes in both A and B, and $Q(n)$ is the complexity of insertion and extraction of an element in a priority queue with n elements.*

Proof The complexity of ordering c edges by the sum of the longest paths to the edge in both A and B is $c \log c$. This gives us the order of updates, where there is no vertex updated twice.

In order to implement the deletion of the sink maximising a criterion maintained (Algorihtm 3), one needs to implement another procedure traversing down the directed graph. This, again, is rather straightforward, with pseudocode exhibited in Appendix A. With some care, one can see:

Claim `popVertexUpdateDownstream` runs in time $O(|\delta|Q(|\delta|) + ||\delta||)$.

Proof sketch: No vertex is updated twice, but there are $|\delta|$ vertices to be updated and $||\delta||$ neighbouring vertices to be checked.

Lemma 2 `topVertex` *runs in time* $O(1)$. `popVertex` *runs in time* $O(|\delta|Q(|\delta|) + ||\delta||)$.

Proof The complexity of `topVertex` is given by the "top" operation of a priority queue [17]. The run-time complexity of `popVertex` is dominated by the complexity of `popVertexUpdateDownstream`, which is $O(|\delta|Q(|\delta|) + ||\delta||)$.

Notice that for melding priority queues [17], the amortised complexity of insertion and extraction is $Q(n) := \sqrt{\log \log n}$. Both $|\delta|$ and $||\delta||$ can be large in the worst-case, notably linear in the number of vertices and edges in `merge`. This is, however, perhaps inevitable. In the closely related problem of prioritising vertices of a DAG, where each vertex is assigned a priority such that, for each oriented edge (v, w), priority(v) $<$ priority(w), Ramalingam and Reps [23] have shown a lower bound of $\Omega(n \log n)$ operations on the insertion of m edges in a graph on n vertices. There are good reasons [1] to believe that the performance is considerably better in expectation. Also, notice that the `topVertex` runs in constant time, so that the execution of the scheduler need not hinder the execution of the jobs on the accelerator.

4 Conclusions and Open Problems

As far as we are aware, we have presented the first fully-dynamic data structures for implementing certain out-of-order taskgraph scheduling policies, with an application in the design of drivers for OpenCL accelerators. Unlike the present best data structures for similar operations on (undirected) trees, such as top trees [2], we do not perform updates lazily, which may well leave space for improvement:

- Can top trees be extended to directed acyclic graphs? Top trees are, in turn, based on data structures proposed by Tarjan et al. [8, 26] earlier. Could those be extended?
- Are there lower bounds on the complexity of `popVertex` and `merge`?

There are also great many questions related to the scheduling policies left open:

- What is the broadest class of graphs for which the studied static scheduling policies are asymptotically optimal? This seems to be one of the most important open questions in scheduling.
- What are the limits of stability in realistic models of queuing networks with depedencies? Consider jobs partitioned into groups. How does the stability threshold change, when there are no intra-group dependencies and jobs within each share the dependencies? How does the stability threshold change, when the probability of dependence of group a on a group b is inversely proportional to the difference in their release dates $r_a - r_b$?
- How much could one be benefit from pilot runs improving the run-time estimates? Could we decide how many pilot runs to perform, based on some measures of quality of the estimates obtained so far?
- What are the benefits of dynamic scheduling policies, such as the multi-mode multi-armed bandits [28, 18]? Notably, could they be used to integrate power management considerations? Could the data structure be extended to accommodate such "indices"?
- What are the benefits of "taskgraph prediction"? In a rather different setting, Chekuri et al [6] have shown that the longest path scheduling based on the known dependency graph is not optimal, when we expect changes of the dependency tree in the future and can make educated guesses about their nature.

There are also several questions specific to accelerators developed by ARM. Considering the results on stability referenced in Section 2.3, however, whatever improved policies there might be, it seems highly likely that they will be require the maintenance of weighted long paths. The research into the dynamic data structures implementing them will hence remain highly relevant.

Acknowledgments The work has been supported by the Industrial Mathematics Internships Programme, and funded by EPSRC and ARM Ltd. OpenCL is a trademark of Apple Inc. used by Khronos by permission.

References

1. Ajwani, D., Friedrich, T.: Average-case analysis of online topological ordering. In: Algorithms and computation, *Lecture Notes in Comput. Sci.*, vol. 4835, pp. 464–475. Springer, Berlin (2007). DOI 10.1007/978-3-540-77120-3_41. URL `http://dx.doi.org/10.1007/978-3-540-77120-3_41`
2. Alstrup, S., Holm, J., Thorup, M., de Lichtenberg, K.: Maintaining information in fully dynamic trees with top trees. ACM Trans. Algorithms **1**(2), 243–264 (2005). DOI 10.1145/1103963.1103966
3. Benini, L., Lombardi, M., Milano, M., Ruggiero, M.: Optimal resource allocation and scheduling forthecell be platform. Ann. Oper. Res. **184**, 51–77 (2011). 10.1007/s10479-010-0718-x
4. Bruno, J.: On scheduling tasks with exponential service times and in-tree precedence constraints. Acta Inform. **22**(2), 139–148 (1985). DOI 10.1007/BF00264227
5. Bulterman, R.W., van der Sommen, F.W., Zwaan, G., Verhoeff, T., van Gasteren, A.J.M., Feijen, W.H.J.: On computing a longest path in a tree. Inform. Process. Lett. **81**(2), 93–96 (2002). DOI 10.1016/S0020-0190(01)00198-3
6. Chekuri, C., Johnson, R., Motwani, R., Natarajan, B., Rau, B.R., Schlansker, M.S.: Profile-driven instruction level parallel scheduling with application to super blocks. In: MICRO, pp. 58–67 (1996)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. Second edn. MIT Press, Cambridge, MA (2001)
8. Goldberg, A.V., Grigoriadis, M.D., Tarjan, R.E.: Use of dynamic trees in a network simplex algorithm for the maximum flow problem. Math. Programming **50**(3, (Ser. A)), 277–290 (1991). DOI 10.1007/BF01594940
9. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972), pp. 85–103. Plenum, New York (1972)
10. Katriel, I., Bodlaender, H.L.: Online topological ordering. ACM Trans. Algorithms **2**(3), 364–379 (2006). DOI 10.1145/1159892.1159896
11. Katriel, I., Michel, L., Van Hentenryck, P.: Maintaining longest paths incrementally. Constraints **10**(2), 159–183 (2005). DOI 10.1007/s10601-005-0554-9
12. Kelley Jr., J.E.: Critical-path planning and scheduling: mathematical basis. Operations Res. **9**, 296–320 (1961)
13. Liu, H.F., Chao, K.M.: A tight analysis of the Katriel-Bodlaender algorithm for online topological ordering. Theoret. Comput. Sci. **389**(1-2), 182–189 (2007). DOI 10.1016/j.tcs.2007.08.009
14. Liu, Z., Sanlaville, E.: Preemptive scheduling with variable profile, precedence constraints and due dates. Discrete Appl. Math. **58**(3), 253–280 (1995). DOI 10.1016/0166-218X(93)E0151-N
15. Liu, Z., Sanlaville, E.: Stochastic scheduling with variable profile and precedence constraints. SIAM J. Comput. **26**(1), 173–187 (1997). DOI 10.1137/S0097539791218949
16. Mareček, J., friends: The complexity of scheduling in OpenCL accelerators. Submitted
17. Mendelson, R., Tarjan, R.E., Thorup, M., Zwick, U.: Melding priority queues. ACM Trans. Algorithms **2**(4), 535–556 (2006). DOI 10.1145/1198513.1198517

18. Niño-Mora, J.: An index policy for multiarmed multimode restless bandits. In: ValueTools '08: Proceedings of the 3rd International Conference on Performance Evaluation Methodologies and Tools, pp. 1–6. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2008). DOI http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4410
19. Papadimitriou, C.H., Tsitsiklis, J.: Intractable problems in control theory. SIAM J. Control Optim. **24**(4), 639–654 (1986). DOI 10.1137/0324038
20. Papadimitriou, C.H., Tsitsiklis, J.N.: On stochastic scheduling with in-tree precedence constraints. SIAM J. Comput. **16**(1), 1–6 (1987). DOI 10.1137/0216001
21. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of optimal queuing network control. Math. Oper. Res. **24**(2), 293–305 (1999). DOI 10.1287/moor.24.2.293
22. Pinedo, M., Weiss, G.: Scheduling jobs with exponentially distributed processing times and intree precedence constraints on two parallel machines. Oper. Res. **33**(6), 1381–1388 (1985). DOI 10.1287/opre.33.6.1381
23. Ramalingam, G., Reps, T.: On competitive on-line algorithms for the dynamic priority-ordering problem. Inform. Process. Lett. **51**(3), 155–161 (1994). DOI 10.1016/0020-0190(94)00080-8
24. Ramalingam, G., Reps, T.: On the computational complexity of dynamic graph problems. Theoret. Comput. Sci. **158**(1-2), 233–277 (1996). DOI 10.1016/0304-3975(95)00079-8
25. Stolyar, A., Yudovina, E.: Systems with large flexible server pools: Instability of "natural" load balancing. ArXiv e-print (2010)
26. Tarjan, R.E.: Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. Math. Programming **78**(2, Ser. B), 169–177 (1997). DOI 10.1016/S0025-5610(97)00015-4
27. Tsitsiklis, J.N., Papadimitriou, C.H., Humblet, P.: The performance of a precedence-based queueing discipline. J. Assoc. Comput. Mach. **33**(3), 593–602 (1986). DOI 10.1145/5925.5936
28. Weber, R.R.: Comments on: Dynamic priority allocation via restless bandit marginal productivity indices. TOP **15**(2), 211–216 (2007). DOI 10.1007/s11750-007-0029-9

A Additional Material

Algorithm 4 `insertEdgeUpdateUpstream`$(G_L,\, u \to v)$

Input: Structure G_L, edge $u \to v$ to be added
Output: List of updated vertices U
Effect: Updated structure G_L
$U = \emptyset$
$Q =$ queue$\langle$ WeightedLongestFirst $\rangle(v)$ // forwards
while $Q \neq \emptyset$ **do**
    $v =$pop(Q)
    for $u \in pred(G_L, v)$ **do**
        if level$(u) <$ level(v)+weight(v) **then**
            level$(u) =$ level(v)+weight(v)
            insert(Q, u)
            insert(U, u)
        end if
    end for
end while
return U

Algorithm 5 `insertEdgeUpdateDownstream`(G_L, $u \to v$)

```
Input: Structure G_L, edge u → v to be added
Output: List of updated vertices U
Effect: Updated structure G_L
U = ∅
if l(u) + d(u, v) > l(v) then
    Q = queue⟨ WeightedLongestFirst ⟩() // forwards
    B = queue⟨ WeightedLongestFirst ⟩() // backwards
    insert(Q, ⟨l(v), v⟩)
    while Q ≠ ∅ do
        a = extractMin(Q) // a for affected
        l(a) = max_{x∈pred(G,a)} l(x)+weight(a)
        G_L = G_L \ {u → a | u → a ∈ G_L}
        G_L = G_L ∪ {u → a | x ∈ pred(G, a) ∧ l(u)+weight(a) = l(a)}
        if |succ(G, a)| = 0 then
            level(a) = 0
            insert(B, a)
            insert(U, u)
        end if
        for b ∈ succ(G, a) do
            if l(a)+weight(b) > l(b) then
                insert(Q, ⟨l(b), b⟩)
            else
                if l(a)+weight(b) = l(b) then
                    G_L = G_L ∪ {a → b}
                end if
            end if
        end for
    end while
    while B ≠ ∅ do
        b = extractMin(B) // b for backwards
        for a ∈ pred(G, b) do
            if level(b) < level(a)+weight(a) then
                level(b) = level(a)+weight(a)
                insert(R, b)
                insert(U, u)
            end if
        end for
    end while
else
    if l(u)+weight(v) = l(v) then
        G_L = G_L ∪ {u → v}
    end if
end if
return U
```

Algorithm 6 `computeAffected`$(G,\, G_L,\, w)$

```
Input: Digraph G = (V, E), structure G_L, vertex w to be removed
Output: List A of affected vertices
Q = {w}
A = ∅
while Q ≠ ∅ do
   u = dequeue(Q)
   A = A ∪ {u}
   for v ∈ succ(G_L, u) do
      G_L = G_L \ {u → v}
      if pred(G_L, v) = ∅ then
         insert(Q, v)
      end if
   end for
end while
return A
```

Algorithm 7 `popVertexUpdateDownstream`$(G_L,\, u \to v)$

```
Input: Digraph G = (V, E), structure G_L, edge u → v to be removed
Output: List of updated vertices U
Effect: Structure G_UL is updated
U = ∅
if u → v ∈ G_L then
   G_L = G_L \ {u → v}
   if pred(G_L, v) = ∅ then
      A = computeAffected(G_L, v) // a for affected
      for a ∈ A do
         indeg(a) = |pred(G, a) ∩ A|
      end for
      Q = queue⟨ WeightedLongestFirst ⟩()
      Q = {a ∈ A | indeg(a) = 0}
      while Q ≠ ∅ do
         q = dequeue(Q)
         l(q) = max_{p∈pred(G,q)} l(p)+weight(q)
         G_L = G_L ∪ {p → q | p ∈ pred(G, q) ∧ l(p)+weight(q) = l(q)}
         for a ∈ succ(G, q) ∩ A do
            indeg(a) = indeg(a) − 1
            if indeg(a) = 0 then
               insert(Q, a)
            end if
         end for
      end while
   end if
end if
return U
```