A Unified Online Directed Acyclic Graph Flow Manager for Multicore Schedulers

Karim Kanoun†, David Atienza†, Nicholas Mastronarde‡, and Mihaela van der Schaar⋆

† Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland. ‡ State University of New York at Buffalo (UB), U.S.A. ⋆ University of California, Los Angeles (UCLA), U.S.A
email: {karim.kanoun, david.atienza}@epfl.ch, nmastron@buffalo.edu, mihaela@ee.ucla.edu.

Abstract— Numerous Directed-Acyclic Graph (DAG) schedulers have been developed to improve the energy efficiency of various multi-core systems. However, the DAG monitoring modules proposed by these schedulers make a priori assumptions about the workload and relationship between the task dependencies. Thus, schedulers are limited to work on a limited subset of DAG models. To address this problem, we propose a unified online DAG monitoring solution independent from the connected scheduler and able to handle all possible DAG models. Our novel low-complexity solution processes online the DAG of the application and provides relevant information about each task that can be used by any scheduler connected to it. Using H.264/AVC video decoding as an illustrative application and multiple configurations of complex synthetic DAGs, we demonstrate that our solution connected to an external simple energy-efficient scheduler is able to achieve significant improvements in energy-efficiency and deadline miss rates compared to existing approaches.

I. Introduction

Emerging real-time video processing applications such as video data mining, video search, and streaming multimedia (e.g., H.264 video streaming [17]) have stringent delay constraints, complex Directed Acyclic Graph (DAG) dependencies among tasks, time-varying and stochastic workloads, and are highly demanding in terms of parallel data computation. Multimedia applications are in general modeled with DAGs where each node denotes a task, each edge from node j to node k indicates that task k depends on task j and each group of tasks has a common deadline d_i. As illustrated in Fig. 1, DAG models for applications with dependent tasks can be roughly classified into 4 types depending on the relationship between the task dependencies and task deadlines.

We define a DAG monitoring solution as the module used to process and analyze these DAGs before scheduling the tasks. This module is different from the scheduler and it is responsible for finding parallelization opportunities, tracking the execution of the DAG and providing relevant information to a connected external scheduler. Numerous offline [15][16] and online [6][7][8] DAG monitoring solutions have been proposed to assist schedulers for multimedia applications. However, these solutions are usually closely related to their connected schedulers, and their output cannot be directly exploited by other schedulers. Thus, the problem of finding a generic online DAG monitoring solution to assist online energy-efficient schedulers, making the DAG processing problem independent from the connected scheduler, is becoming increasingly important.

Moreover, we also contend that none of these existing online DAG monitoring solutions have considered the general DAG model in which a task's children can have different deadlines (e.g., model 4 in Fig. 1). We present an example of this problem in the remainder of this paragraph. Fig. 2 illustrates an example of two different DAGs modeling the same H.264 video decoder application. The DAG of Fig. 2a preserves the original dependencies between I, P and B frames, where I-frames are compressed independently of the other frames, P-frames are predicted from previous frames, and B-frames are predicted from previous and future frames [1]. Each frame is composed of three type of tasks, namely, initialization (e.g., I_1), slice decoding (e.g., S_2, S_3 and S_4) and the deblocking filter (e.g., F_5). In the example shown in Fig. 2 with 4 frames (I-B-P-B), there are 2 deadlines corresponding to the display deadlines of the two B frames. These deadlines are imposed by the frame rate and the underlying dependency structure. In fact, if frame k depends on frame $k+l$ (with $l > 0$), then both frames will have their deadlines set to the minimum one, i.e., $k/30$ seconds. Finally, in this DAG model, a task's children may have different deadlines (e.g., $F_5 \rightarrow (I_6, I_{16})$). Existing online DAG monitoring approaches (cf. Section II) are not able to handle the additional dependencies between the tasks with different deadlines. Instead, they are forced to convert the original DAG to the fork-join DAG model as presented in Fig. 2b where critical edges (i.e., edges linking 2 tasks with different deadlines) are removed and replaced by a single join edge that links the last task with deadline d_i to the first task with deadline d_{i+1}. Although the fork-join model (which is defined as a sequence of sequential and parallel segments [2]) preserves the dependency coherency between tasks, it restricts the scheduler to operate on tasks belonging to one deadline at a time (i.e., the earliest deadline). Hence, several parallelization opportunities are missed (e.g., frame B with a deadline d_i and frame P with a deadline d_{i+1} can be decoded in parallel, once frame I is decoded).

To summarize, each existing scheduler implements its own DAG monitoring solution with several restrictions on the DAG model. Moreover, none of the existing solutions are able to handle the general DAG illustrated with model 4 in Fig. 1, which allows a task's children to have different deadlines.

In this paper, we propose a novel unified DAG monitoring solution, which we call DAG Flow Manager (DFM). The key contributions of this work are as follows:

This work was supported in part by a Joint Research Grant for ESL-EPFL by CSEM, and the Spanish Government Research Grant TIN2008-00508.

978-1-4799-2816-3/14/$31.00 ©2014 IEEE

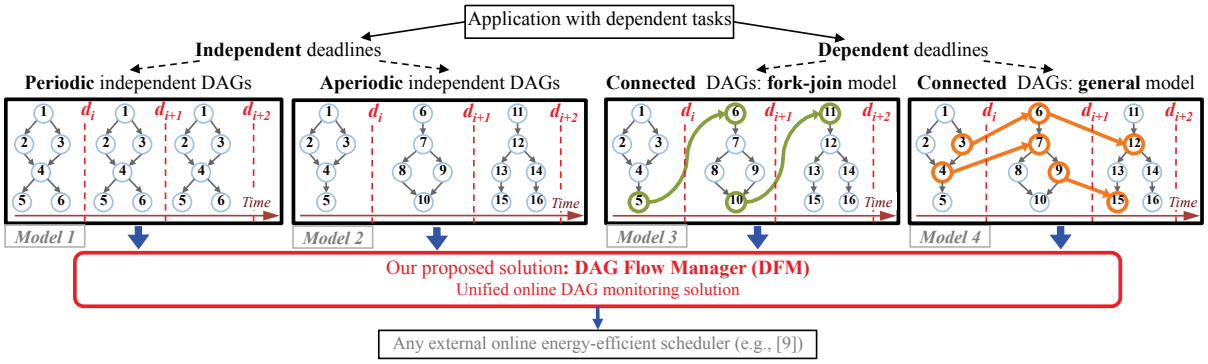
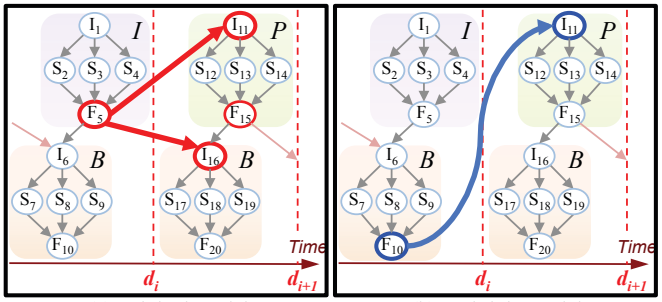


Fig. 1. Classification of DAG models of applications with dependent tasks. Our proposed solution handles all possible DAG models



(a) Original model (b) Fork-join model

Fig. 2. H.264 decoder DAG model [1]

• A low-complexity online DAG monitoring solution that is fully independent of the scheduler that it is connected to it.

• Our DFM does not impose any restrictions on the DAG (e.g., restrictions on deadline dependencies as in the fork-join model). Our DFM covers online all DAG models (Fig. 1).

• Our DFM provides detailed information about the execution status of tasks and deadlines within a look-ahead window, allowing simple connected schedulers to have optimal control of the core assignment and DVFS selection of each task.

Our results for the H.264 video decoder and different configurations of synthetic DAGs demonstrate that our proposed DFM allows a connected online scheduler based on [9] to reach over 50% reduction in energy consumption and over 80% reduction in deadline miss rates compared to existing DAG monitoring solutions [7][8] connected to the same scheduler.

The remainder of this paper is organized as follows. In Section II, we describe the limitations of current offline and online DAG monitoring approaches. In Section A, we introduce the system and application model. In Sections B, C, D and E, we describe our online DFM algorithm in detail. In Section IV, we present our experimental results. Finally, we summarize the main conclusions in Section V.

II. Related work

In Table I, we summarize different application models considered by existing DAG analyzers and we compare them to the DAG modeled by our solution.

Existing static approaches [15][16] model the application as a DAG with periodic dependent tasks as shown in Fig. 1 (model 1). They propose a coarse-grained task-level software pipelining algorithm to transform periodic dependent tasks into a set of independent tasks based on a retiming technique. However, these approaches are unsuitable for multimedia applications with dynamic DAGs. In fact, the assumption of a periodic DAG limits the applicability of static approaches because highly optimized modern and emerging video coders do not always have periodic task-graphs (e.g., they may use adaptive group of pictures structures). Hence, applying techniques such as pipelining is not possible, especially for applications that do not adopt a fixed task-graph structure but instead adapt their task-graphs on the fly (e.g., stream mining applications [18]). Moreover, for H.264 video decoding, such pipelining techniques require buffering delays that are proportional to the Group of Pictures (GOP) size, which may be large. Finally, static approaches rely on worst-case execution time estimates to generate the schedulers input. These approaches are efficient if the workload and the starting time of each task is fixed and known. However, they are unsuitable for multimedia applications with dynamic workloads because modeling a non-deterministic workload with its worst-case execution time leads a connected scheduler to create significant slack time and utilize resource inefficiently.

Few online DAG monitoring solutions [6][7][8] have been recently proposed for scheduling problems. In [6][7][8], they consider periodic tasks where each task is represented as an independent DAG with a single deadline (i.e., a task is modeled as a group of jobs or threads having the same deadline similar to models 1-2 of Fig. 1). Therefore, in this approach, monitoring n deadlines simultaneously requires n independent DAGs. Then, each DAG (i.e., each deadline) is decomposed into segments in order to identify future parallelization opportunities. Therefore, if we consider the general case of DAG of applications with dependent deadlines illustrated with model 4 of Fig. 1 and Fig. 2a where a job's children may have different deadlines, the solutions presented in [7][8] are then forced to convert these DAGs into the fork-join model (e.g., model 3 of Fig. 1 and Fig. 2b). Although the fork-join model preserves the dependencies between tasks, it restricts schedulers to scheduling tasks belonging to only one deadline at a time. Hence, several parallelization opportunities are missed (e.g., Fig. 2: frames B and P with deadline d_i and d_{i+1} respectively). The solu-

TABLE I
Comparison of existing DAG monitoring approaches to our DFM: Application model and analysis type

DAG-monitoring solution	type of deadlines	type of DAG	analysis
[15, 16]	independent	periodic	offline
[6, 7, 8]	independent	periodic	online
[13]	dependent	H.264	online
Our solution	**dependent**	**general**	**online**

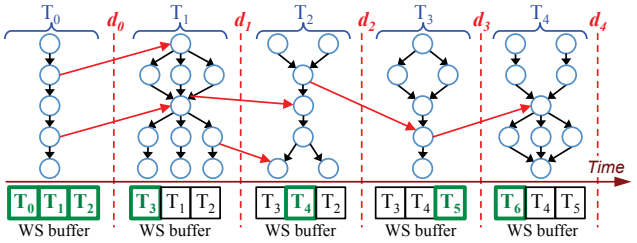


Fig. 3. Evolution of the WS buffer during the execution

tions in [10][9] also suffer from the same limitation. Only one existing solution accounts for dependencies among tasks with different deadlines [13], however, it is restricted to the H.264 DAG model.

III. Online DAG Flow Manager (DFM)

A. System Model

We model our target computationally intensive application as a DAG $G =< \mathcal{N}, \mathcal{E} >$ of dependent tasks t_j with non-deterministic workload w_j and coarse-grained deadlines. $\mathcal{N}$ is the node set containing all the tasks. $\mathcal{E}$ is the edge set, which models the dependencies between the tasks. Each node in the DAG denotes a task t_j. e_j^k denotes that an edge is pointing from t_j to t_k indicating that task k depends on task j. Each task t_j is characterized with its index j and a deadline d_i (i is the deadline index). Each deadline can be assigned to a subset of tasks. Our model covers all general DAG models (e.g., Fig. 1) including the general case where a task's children may have different deadlines than the task itself and its other children (e.g., model 4 of Fig. 1). Finally, we assume that our target multicore platform has M cores with DVFS capability to trade off energy consumption and delay. Each processor can operate at a different frequency $f_i \in F$, where F denotes the set of available operating frequencies and $f_i < f_{i+1}$.

B. Overview of the proposed DFM

We define T_i as the subset of tasks having the same deadline d_i. We also define the Working Set WS as a look-ahead window buffer of n T_is. Each time all the tasks inside a T_i finish their execution, we request the next T_i input from the application. For each new added T_i to the WS buffer, our approach requires from the application its adjacency matrix, its deadline value and the list of edges connecting this T_i with T_{i+l} (with $l \neq 0$). In our *DAG Flow Manager (DFM)*, we propose then to process the full DAG of the application using this WS buffer where only a limited number of deadlines are processed at a time. Analyzing the full DAG of an application by subset of n deadlines (i.e., T_is) is the key for having a low complexity online DAG monitoring solution. Thus, it will be possible to efficiently adapt to applications that have a highly variable workload and adapt their task-graphs on the fly. Fig. 3 illustrates an example of our DFM processing a general DAG model using a WS buffer of 3 deadline slots. In this example, for the sake of clarity, we assume that each T_i finishes its execution at d_i. First, at the beginning of the execution, the WS buffer is filled with T_0, T_1 and T_2. Next, when a deadline finishes its execution, a slot becomes available and it gets automatically filled with the next available deadline. In Section C, we describe the initialization phase that we apply on each new added T_i to the WS buffer and how we handle online the additional dependencies between these T_is. In Section D, we then describe different data structures that our DFM provides to assist an external scheduler and how they can be used efficiently. Finally, we explain how we update online the generated data structures and synchronize it with recent scheduling decisions.

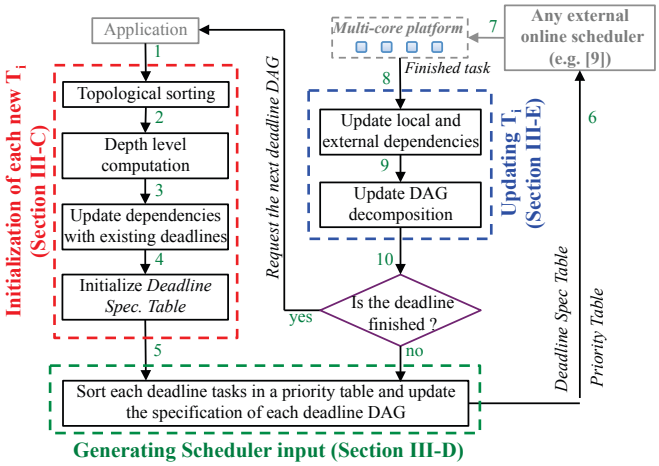


Fig. 4. A complete overview of our DFM

C. Initialization of each new added deadline to the WS

The full initialization process is illustrated in Fig. 4 (arrows 1, 2, 3 and 4). First, we sort the tasks of each new added T_i in the WS buffer, in topological order. An optimal topological sorting algorithm with a complexity of $\mathcal{O}(|\mathcal{N}| + |\mathcal{E}|)$ has been proposed in [12]. Moreover, in [14], it was demonstrated that a DAG with n nodes has a worst case of $n * (n-1)/2$ edges. The topological sorting step is applied only once for each new T_i. This reduces the complexity of computing the depth δ_i^j of each task t_j with deadline d_i in the WS buffer. While sorting the tasks, the list of direct parent tasks $\mathcal{L}_j^P$ (i.e., list of tasks t_k linked to ingoing edges e_k^j) and the list of direct children tasks $\mathcal{L}_j^C$ (i.e., list of tasks t_k linked to outgoing edges e_j^k) for each task t_j are also generated. $\mathcal{L}_j^P$ is used to compute the critical path workload in Section D, while $\mathcal{L}_j^C$ is used for the depth level computations of each available task in the WS buffer. In fact, for each outgoing edge e_j^k (i.e., edges connecting t_j to $\mathcal{L}_j^C$ tasks) of each remaining task t_j in T_i, our solution updates the depth value δ_i^k of the node t_k with $\delta_i^k \leftarrow max(\delta_i^j + 1, \delta_i^k)$. The complexity of the graph traversal algorithm that we apply to compute δ_i^k is then $\mathcal{O}(|\mathcal{E}|)$.

We denote by $l_{i,k}$ the group of tasks t_j having the same depth level $\delta_i^j = k$ in T_i. Note that, for all tasks in T_i, tasks at depth level $k+1$ (i.e., $l_{i,k+1}$) can only be scheduled after tasks at depth level k (i.e., $l_{i,k}$) are finished. Fig. 5 illustrates in detail the difference between t_j, T_i, $l_{i,k}$ and δ_i^j after applying our algorithm on a WS buffer containing part of the DAG of the H.264 video decoder.

While traversing the DAG for the first time for each new T_i, the number of unfinished direct parent tasks t_k for each task t_j, that we call the dependency status r_j, is also computed. r_j is used to assist the scheduler in detecting available parallelization opportunities from the remaining unscheduled tasks in the WS (i.e., entry nodes $r_j = 0$). In fact, the dependency status r_j is the key idea to track existing dependencies between tasks

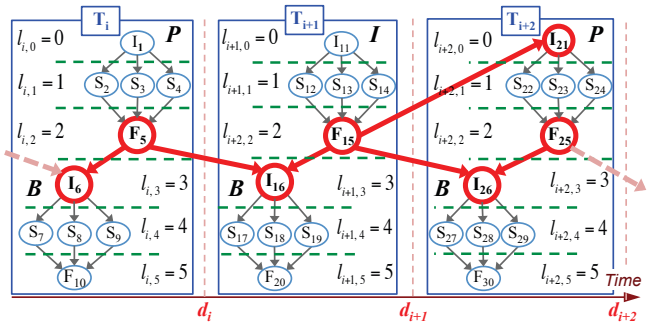


Fig. 5. The decomposition applied by our DFM on H.264 DAG [1]

with different deadlines (i.e., between T_is). Once the initial dependency status r_j is computed for each task within the newly added T_i, we check for its dependencies with other deadlines. To this end, for each task t_j in T_i with an incoming edge from tasks with earlier deadlines (i.e., T_{i+l} with $l < 0$), we increment its r_j value. Then, for each task t_j in T_i with an outgoing edge to other deadlines tasks (i.e., T_{i+l} with $l > 0$) we update its list of direct children tasks $\mathcal{L}_j^C$. This will be used later for the update phase in Section E for clearing the dependencies between the T_is. Finally, it may happen that a task finishes its execution and the other deadline that is connected to does not yet exist in the WS buffer. In this case, we need to make sure that when this new deadline is added to the buffer, its dependency with this finished task is cleared correctly. Therefore, we store this dependency in a new list, that we call the *list of non-cleared dependencies*. This list is generated during the update process (cf. Section E). We use the list of non-cleared dependencies in the last step of this initialization phase in order to check if a dependency has to be cleared by decrementing the r_j value of the concerned task. Each time a dependency is cleared, we remove its corresponding entry from the list.

D. Generation of the scheduler input

To assist online schedulers with immediate parallelization opportunities and the priorities of the available tasks, we define the first output structure, called *Priority Table*, that we provide to any connected scheduler. Each entry in the Priority Table corresponds to a task t_j and it is characterized by: estimated workload w_j, earliest release time s_j, expected ending time z_j, fixed deadline d_i^j where i refers to T_i and $d_i^j = d_i \ \forall j$, critical path workload to its deadline $w_j^{d_i}$ and the dependency status r_j (i.e., the total number of parent tasks that it still depends on). w_j, s_j, z_j and $w_j^{d_i}$ are in clock cycles and d_i^j is in seconds. The tasks in the Priority Table are sorted by our DFM according first to their deadline d_i, then refined to their depth level $l_{i,k}$ in T_i and finally to their estimated workload in case of a tie. We use the Quicksort algorithm with an average complexity of $\mathcal{O}(|\mathcal{N}|.log(|\mathcal{N}|))$. Several workload estimation methods [3][4] with a negligible overhead have been proposed for multimedia applications. Full overhead measurements on the H.264 decoder application are provided in Section IV.

Regarding the computation of this output, z_j and s_j are calculated while the WS buffer is traversed by the depth level computation algorithm (cf. Section C) with $z_j \leftarrow w_j + s_j$ and $s_k \leftarrow max(z_j, s_k)$ for $\forall t_k \in \mathcal{L}_j^C$. If the task is currently running at frequency f_j then we use $z_j \leftarrow w_j * \frac{f_{max}}{f_j} + s_j$ to take into account the applied frequency. Finally, for the calculation of the critical path workload value $w_j^{d_i}$ starting from each task t_j to its local sink node in T_i, the method used to calculate $w_j^{d_i}$ is the same as the depth level computation algorithm except that we traverse the topological graph in the inverse order and we replace $\mathcal{L}_j^C$ with $\mathcal{L}_j^P$. Our solution indicates to the scheduler which tasks are entry nodes in the remaining tasks set using the dependency status r_j of each task t_j. Nodes with $r_j = 0$ in the Priority Table are potential starting tasks for parallelization. Even though the Priority Table gives a straightforward solution to select the task to schedule, it does not give the scheduler general information related to the execution status of each remaining T_i in the buffer, such as their currently running tasks progress. Having such information, a scheduler will be able to efficiently tune the deadlines values for a more energy efficient execution [9]. Moreover, such information could be exploited by any connected scheduler to set the priority of each T_i, which can be efficiently used in parallel with the priority table. To this end, we then propose a second output to online schedulers in order to track the execution status of future deadlines tasks that we call the *DeadlineSpec Table*. This latter output is used to track the overall progress of each group of tasks T_i. Each entry in the DeadlineSpec Table stores relevant computed information related to each T_i namely: total workload, executed workload, scheduled workload and finally a depth table. Each entry k in the depth table corresponds to a $l_{i,k}$ in T_i and it stores relevant computed information namely: total workload, maximum number of allowed cores and minimum amount of parallelizable workload. The full DeadlineSpec Table is initially generated using previously computed information related to each task. The complexity of creating this table is then linear to the number of nodes in the considered WS.

E. Updating the DAG decomposition and the scheduler input

As shown in Fig. 4 (arrows 8, 9 and 10), the working set decomposition and the generated output are updated each time a task finishes its execution. First, we remove the finished task t_j from the list of remaining tasks and re-estimate the workload of similar tasks. Then, we decrement r_k of each task t_k in $\mathcal{L}_j^C$ (i.e., its direct children tasks t_k). However, it may happen that one direct child has a different deadline that is not available yet in the WS buffer. In this case, we save this dependency in a list that contains all the non-cleared dependencies. This list is used during the initialization phase to clear future dependencies when a new T_i linked to this edge is added to the WS buffer as described in Section C. This first step allows instantly detecting new entry nodes (i.e., when $r_j = 0$) among all the available tasks of all the T_is in the Priority Table output. Then, we set the earliest starting time of each task t_j depending on its execution status. If the task t_j is an entry node and did not start yet, then the current execution time is assigned as its earliest starting time. However, if the task t_j is currently executing then some values from the DeadlineSpec Table, namely, the scheduled workload and executed workload, are updated from values of the currently running task t_j. Finally, we update the information related to each remaining task t_j in T_i by applying the same graph traversal algorithm of Section C on the remaining unscheduled nodes of T_i. Therefore, the total number of times the graph traversal algorithm is applied for the full execution of a T_i will be equal to twice the number of tasks in the

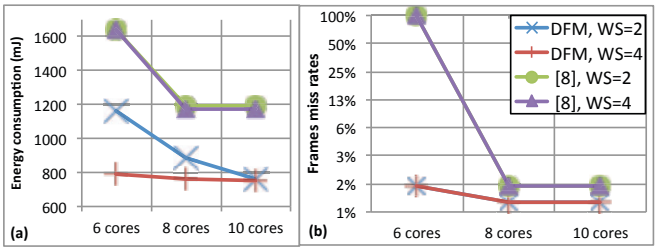


Fig. 6. H.264 decoder: comparison between our DFM and [8], connected to the same scheduler. (a)Energy consumption. (b)Deadline miss rates.

T_i and during each update it will be executed with one node less (i.e., without the finished tasks). Indeed, each time a task finishes its execution the algorithm will be executed first during the depth computation phase and then for the critical path workload. This guarantees real-time information regarding future available parallelization each time before a scheduler has to make a decision.

IV. Experimental Results

We demonstrate the advantages of our DFM, which we have implemented in C, to assist an energy-efficient online scheduler on two experimental benchmarks namely, the H.264 video decoder [17], and multiple configurations of synthetic DAGs generated with GGEN tool [11]. We explain our experimental setup in Section A. Then, we present our results for the two sets of experiments in Sections B and C.

A. Experimental setup

As described in Sections I and II, existing online DAG monitoring solutions [6][7][8] do not consider DAG models where a task's children can have different deadlines. The only way for these approaches to handle such a DAG online is to process it deadline by deadline (i.e., fork-join parallelism model) where critical dependencies between the deadlines are replaced with a single join edge as shown in Fig. 2. However and since our DFM applies similar task decompositions technique to existing solutions for DAGs without critical dependencies between the deadlines, we can perfectly simulate existing approaches by first converting general DAG model to the fork-join model then by applying our DFM on the converted DAGs.

We connect the simulated existing solution [8] ([8] uses the same decomposition technique as in [7]) and our DFM to an external online scheduler based on [9], which applies the least possible restrictions on its application model compared to other schedulers. In [9], an online scheduling approach called MLTF was proposed for multimedia application, where the earliest deadline is scheduled with limited consideration of future tasks' deadlines and workloads. Indeed, based on the derived estimated duration of all pending tasks, a new virtual deadline is set in order to have more balanced workload distribution over the time. Then, a Largest Task First (LTF) schedule is applied. In [9], the algorithm applied for frequency selection does not take into account the dependencies between the tasks. Therefore, we made the scheduler select the frequency of each task based on the remaining critical path workload and the available amount of clock cycles. Due to dependencies between tasks, gaps (i.e., when a core is idle and waiting for another task to finish) may occur during the schedule. Therefore, we add another simple module on top of the MLTF scheduler presented in [9] to fill the gap. This module compares the amount of available gap (which can be easily estimated from the schedule generated by [9]) to the workload estimation of the available task t_j with $t_j \in T_{e+l}$ (i.e., having deadline d_{e+l}) to compute the minimum frequency $f^j_{d_e}$ that allows t_j to fit the available gap occurring before d_e. Then, the module applies again the MLTF schedule but this time on T_{e+l} between d_{e+l} and d_{e+l-1} to estimate the minimum frequency $f^j_{d_{e+l}}$ used when scheduling task t_j during its allocated time (i.e., between d_{e+l} and d_{e+l-1}. Finally, if $f^j_{d_e} \leq f^j_{d_{e+l}}$, then the gap is filled. All these computations are then based on MLTF [9].

B. H.264 decoder - Energy, deadline miss rates and overhead

For our multimedia benchmark, we have used the Joint Model reference software version 17.2 (JM 17.2) of an H.264 encoder [17]. The DAG model and the deadlines configuration that we consider for our benchmark are similar to the one shown in Fig. 2 with an IBPB GOP structure, 8 slices per frame and 30 frames/second for CIF (352x288) resolution video sequences. We use accurate statistics generated from an H.264 decoder that we have parallelized and executed on a sophisticated multiprocessor virtual platform simulator. In fact, in this work, we use the multiprocessor ARM (MPARM) virtual platform simulator [5], which is a complete SystemC simulation environment for MPSoC architectural design and exploration. MPARM provides cycle-accurate and bus signal-accurate simulation for different processors. In our experiments, we have generated with MPARM the workloads and the dynamic power consumption statistics of each task (i.e., frame initialization, slice decoding and deblocking filter) using ARM9 power consumption figures with DVFS support (300MHZ at 1.07V, 400MHZ at 1.24V and 500MHZ at 1.6V).

In Fig. 6, we show the energy consumption and frame miss rates when decoding the Foreman sequence. We have used the same scheduler connected first to our DFM and second to existing DAG monitoring solution [8] as described in the previous section. The results of Fig. 6a show that [8] does not allow the connected scheduler to use more than 8 cores as the maximum number of parallelizable tasks within a single T_i is 8 in the considered DAG (i.e., 8 slices per frame). However, our solution allows the connected scheduler to take advantage of the additional number of cores as demonstrated by the relative energy decreasing with the number of cores. The results show also that our DFM can efficiently exploit the increased size of the WS buffer. In fact, our DFM allows the connected scheduler to reduce the energy consumption by up to 52% compared to [8] thanks to the information provided by our DFM regarding each of the available deadlines in the buffer to the connected scheduler. For the frame miss rates, the decomposition technique and the information provided by existing DAG monitoring solution do not allow the connected scheduler to efficiently schedule the tasks before their deadlines. In fact, as shown in Fig. 6b, all the frames are missed if the number of cores is less than 8 cores due to heavy workloads. However, by exploiting our DFM output related to each of the available deadlines in the buffer and the detected parallelization among the deadlines tasks, the connected scheduler is then able to achieve less than 1.5% miss rates starting from only 6 cores as the gaps are filled with available tasks with future deadlines.

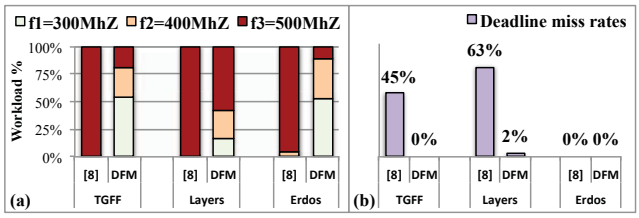


Fig. 7. Synthetic DAGs (deadlines values are only 1% greater than the critical path workload) : comparison between our DFM and [8], connected to the same scheduler. (a)Frequency usage. (b)Deadline miss rates.

Finally, we present the overhead in terms of processor clock cycles with respect to the workload of the Foreman video sequence. We measured an overhead of 0.82%, 0.93% and 1.04% of our proposed DFM when using 2, 3 and 4 deadlines per WS respectively (with 20 tasks per deadline). The overhead of the connected scheduler varies between 0.7% and 1.24% depending on the number of cores and the WS buffer size.

C. Generalizing the results to the general DAG model

We use the GGEN [11] tool to model an application with 100 connected generated DAGs. We connect these DAGs by randomly adding m edges in a way that some tasks in DAG g depend on some other tasks in DAG $g-1$. For the workload, we assume that an application of n tasks has k types of workloads. We assign then each task t_j with a random type number a_j between 1 and k, and we compute the workload with $w_j = w(a_j) + x$ where $w(a_j)$ is the minimum workload value of all the tasks with type a_j, and x is a random number between 0 and $(w(a_j) * \alpha)$ with $\alpha \in [0, 0.5]$. x represents the workload variation of each task with respect to its type. Finally, to assign a deadline to each DAG, we compute the critical path workload w_i^{cp} of each DAG i and the final deadlines values (in seconds) are assigned with $d_i = d_{i-1} + w_i^{cp} * \frac{1+\beta}{f_{max}}$ with $\beta \in [0, 0.5]$.

We generate the synthetics DAGs with Erdos, TGFF and Layers DAG generation methods as presented in [11]. We set the number of cores to 6 and previously described parameters to $n = 25$, $k = 5$, $\beta = 0.01$ (i.e., deadlines values are only 1% greater than the critical path workload), $\alpha = 0.4$ and $m \in [10, 15]$. For TGFF method, we set the maximum number of ingoing and outgoing edges per node to 4, for Erdos and Layer we set the probability of an edge to appear in each DAG to 0.5 and for the Layer method we set the number of layers to 4. We choose these parameters in order to simulate a congested system. Fig. 7a shows the distribution of the frequency usage of the total workload assigned by the scheduler exploiting our DFM output compared to the same scheduler exploiting [8] output. A higher fraction of workload processed at lower frequencies is desirable because it indicates lower dynamic energy consumption. We also compare the deadlines miss rates in Fig. 7b. Our DFM significantly reduces the usage of the maximum frequency by up to 84% and the deadline miss rates by up to 61% (with 0% to 2% miss rates overall). Our solution provides the information related to the execution status of each deadline which allows a connected scheduler to take foresighted decisions that target the lowest possible frequency, reducing then the dynamic energy consumption. Moreover, connecting the deadlines together with multiple edges restricts existing solutions [8] to provide the scheduler with only parallelization within a single deadline. Therefore, our DFM allows any connected scheduler to exploit more parallelization opportunities resulting into a more balanced workload distribution and more optimal usage of the available resources.

V. Conclusion

In this paper we have proposed a novel unified DAG monitoring solution, that we called DAG Flow Manager (DFM). The key contributions of this work were as follows: (i) A low-complexity online DAG monitoring solution that is fully independent of the scheduler that it is connected to it; (ii) Our DFM does not impose any restrictions on the DAG and covers online all DAG models (Fig. 1); (iii) Our DFM provides detailed information about the execution status of tasks and deadlines within a look-ahead window, allowing simple connected schedulers to have optimal control of the core assignment and DVFS selection of each task. Our results for the H.264 decoder have demonstrated that our proposed DFM solution allowed a connected online scheduler to reach up to 52% reduction in energy consumption and over 80% reduction in deadline miss rates compared to the schedule generated by the same scheduler but relying on existing DAG monitoring solutions. The overhead in terms of processor clock cycles for the proposed DFM, for the aforementioned results, is less than 1% with respect to the total workload of the foreman video sequence. Finally we have generalized these results on synthetic DAGs with different DAG configurations.

References

[1] T. Wiegand, *et al.*, "Overview of the h.264/avc video coding standard," in *IEEE TCSVT*, vol. 13, no. 7, pp. 560–576, July 2003.

[2] *"OpenMP," http://openmp.org*.

[3] Y. Andreopoulos, *et al.*, "Adaptive linear prediction for resource estimation of video decoding," in *IEEE TCSVT*, vol. 17, no. 6, pp. 751–764, June 2007.

[4] S.-Y. Bang, *et al.*, "Run-time adaptive workload estimation for dynamic voltage scaling," in *IEEE TCAD*, vol. 28, no. 9, pp.1334 –1347, Sept. 2009.

[5] L. Benini, *et al.*, "Mparm: Exploring the multi-processor soc design space with systemc," *J. VLSI Signal Process. Syst.*, vol. 41, no. 2, pp. 169–182, Sept. 2005.

[6] M. Qamhieh, *et al.*, "A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model," in Proc. *RTAS*, Apr. 2012.

[7] A. Saifullah, *et al.*, "Real-time scheduling of parallel tasks under general dag model," *online: http://www.cse.wustl. edu/˜ saifullaha/BIB_Files/dag_parallel. pdf*, 2012, tech. report.

[8] J. Li, *et al.*, "Outstanding Paper Award: Analysis of Global EDF for Parallel Tasks," in Proc. *ECRTS*, 2013..

[9] Y.-H. Wei, *et al.*, "Energy-efficient real-time scheduling of multimedia tasks on multi-core processors," in Proc. *ACM SAC*, 2010.

[10] J. Cong, *et al.*, "Energy efficient multiprocessor task scheduling under input-dependent variation," In Proc *DATE*, 2009.

[11] D. Cordeiro, *et al.*, "Random graph generation for scheduling simulations," in Proc. *SIMUTools*, 2010.

[12] T. H. Cormen, *et al.*, "*Introduction To Algorithms*," MIT Press, 2001.

[13] N. Mastronarde, *et al.*, "Markov decision process based energy-efficient on-line scheduling for slice- parallel video decoders on multicore systems," in *IEEE TMM*, vol. 15, no. 2, pp. 268–278, Feb, 2013.

[14] O. Sinnen, "Task scheduling for parallel systems," in Wiley, 2007.

[15] Y. Wang, *et al.*, "Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip, in " *ACM TODAES*, vol. 16, no. 2, pp. 14:1–14:32, Apr. 2011.

[16] Y. Wang, *et al.*, "Optimal task scheduling by removing inter-core communication overhead for streaming applications on mpsoc," in Proc. *RTAS*, 2010.

[17] *H.264/14496-10 AVC Reference Software Manual (revised for JM 17.1).*

[18] R. Ducasse, *et al.*, "Adaptive topologic optimization for large-scale stream mining," in *IEEE JSTSP*, vol. 4, no. 3, pp. 620–636, June 2010.