Fast Multiprocessor Scheduling with Fixed Task Binding of Large Scale Industrial Cyber Physical Systems

Shreya Adyanthaya[1], Marc Geilen[1], Twan Basten[1,2], Ramon Schiffelers[3,1], Bart Theelen[2], Jeroen Voeten[2,1]
[1]*Eindhoven University of Technology, Eindhoven, The Netherlands*
[2]*TNO-ESI, Eindhoven, The Netherlands*
[3]*ASML, Veldhoven, The Netherlands*
{*s.adyanthaya, m.c.w.geilen, a.a.basten, j.p.m.voeten*}*@tue.nl, ramon.schiffelers@asml.com, bart.theelen@tno.nl*

***Abstract*—Latest trends in embedded platform architectures show a steady shift from high frequency single core platforms to lower-frequency but highly-parallel execution platforms. Scheduling applications with stringent latency requirements on such multiprocessor platforms is challenging. Our work is motivated by the scheduling challenges faced by ASML, the world's leading provider of wafer scanners. A wafer scanner is a complex cyber-physical system that manipulates silicon wafers with extreme accuracy at high throughput. Typical control applications of the wafer scanner consist of thousands of precedence-constrained tasks with latency requirements. Machines are customized so that precise characteristics of the control applications to be scheduled and the execution platform are only known during machine start-up. This results in large-scale scheduling problems that need to be solved during start-up of the machine under a strict timing constraint on the schedule delivery time. This paper introduces a fast and scalable static-order scheduling approach for applications with stringent latency requirements and a fixed binding on multi-processor platforms. It uses a heuristic that makes scheduling decisions based on a new metric to find feasible schedules that meet timing requirements as quickly as possible and it is shown to be scalable to very large task graphs. The computation of this metric exploits the binding information of the application. The approach will be incorporated into the ASML's latest generation of wafer scanners.**

***Keywords*-Multiprocessor scheduling, Scalability, Latency, Fixed binding, Industrial case study, cyber-physical systems**

I. Introduction

ASML[1] manufactures complex machines that are critical to the production of integrated circuits or chips. These machines manipulate silicon wafers by exposing them to a light source with extreme precision for the production of chips at high production rates. This exposure process is called *lithography*. ASML is the world's leading provider of lithography systems for the semiconductor industry. After the exposure process, the wafers are post-processed and separated into integrated circuits. The operations of the machine's components, such as the movements and exchange of wafers as well as their exposure, are highly complex and require a very large number of control operations to be carried out accurately. Hence, the wafer scanner contains a large number of closed loop or servo control systems. Servo control systems consist of control applications that are mapped onto an execution platform, as depicted in Figure 1. Control applications read values from sensors, compute mathematical functions specified in control tasks, and send results to actuators. Wafer scanners have hundreds of control applications, each with hundreds of control tasks, sensors and actuators and this complexity grows with each new generation of machines. To achieve the required machine performance, applications have to run at high rates and have to satisfy stringent latency requirements between sensing and actuation. For this purpose a lot of computational power is needed, which is offered by multi-processor platforms consisting of tens of general purpose (multi-core) processors communicating through low-latency packet-switched communication networks.

Scheduling the control applications on these platforms in the best possible manner is a prime concern. Schedules that meet latency requirements are called *feasible schedules*. The ASML wafer scanners are customized so that many of the required configuration parameters defining the characteristics of the servo control applications and execution platform are available only on the start-up of the machine. As a result, the schedules need to be produced during machine start-up. The computation time of the schedules contributes to the startup time of machines. Down-time is very costly because these machines process many ($\pm$ 200) wafers per hour, and these wafers are very expensive. Hence, scheduling time needs to be minimal. The scheduling problem is shown to be NP-Complete. In this work, we use a heuristic for list scheduling based on a new metric called the due-date metric, the computation of which exploits the binding information of the application, to rapidly obtain feasible schedules of very large applications with strict latency requirements. Computation of the optimal value for this due-date metric is as hard as the scheduling problem and hence is in itself NP-Complete. This work presents a simple and efficient algorithm for computation of a sufficiently good value of the due-date metric that allows the scheduler to obtain schedules that meet the latency requirements in minimal time. Experiments performed on applications obtained from the wafer scanner control domain allowed us to successfully

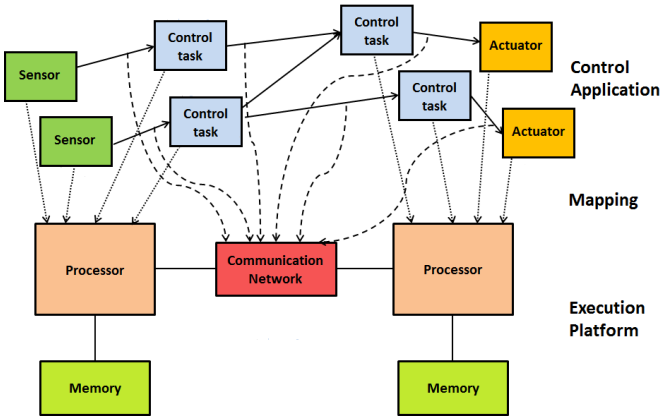

Figure 1: Servo control application mapped on execution platform.

verify the functioning of the scheduler in real industrial scenarios. The schedulers will be incorporated into the latest generation of ASML's machines.

As a motivational example consider a fragment of an example application shown in Figure 2. A task t is defined by an execution time e, a resource r and a deadline d. For instance, task a_8 has an execution time of 8, is bound to resource P_1 and has a deadline 140. A dependency between two tasks implies that the source task needs to be completed before the target task can begin. A task is enabled when all its predecessors have completed their execution. In Figure 2, both tasks a_8 and a_{13} are enabled at the same time. We make a choice of which one of the two must be scheduled sooner on P_1 based on the classical *ALAP* (*CALAP*) completion times of the two. *CALAP* of any task t is the minimum of the ($CALAP_s - e_s$) of each successor s and its own deadline d_t. Although the deadline of a_{25} is 140, its *CALAP* value is 119. This has been computed by propagating backwards from the leaf nodes of the complete graph, most nodes of which have been omitted from Figure 2 for the sake of simplicity. In turn, a_{16}, a_{22} and a_{23}, all having a_{25} as their only successor, derive a *CALAP* value of (*min*(119-10, 140)) = 109. Thereafter, a_8 gets a *CALAP* of 101 and a_{13} gets a *CALAP* of 102. Based on *CALAP* values, a_8 is given a preference over a_{13} and is scheduled earlier. This results in a large gap in the schedule shown in the snapshot of the Gantt chart in Figure 3, produced using special purpose tools. Consequently, a later task in the schedule misses its deadline. This happens because all the tasks dependent on a_{13} have to wait for it to finish execution. While computing the *CALAP* times we overlook the extra information of the task bindings, thus ignoring the fact that a_{13} has higher workload depending on it on the same resource as compared to a_8. This information is lost in the parallelism of its successors a_{22} and a_{23} which eventually will be scheduled sequentially on resource P_2. This binding information can be taken into account in the computation of a new bound on the task completion time called the due-date denoted by

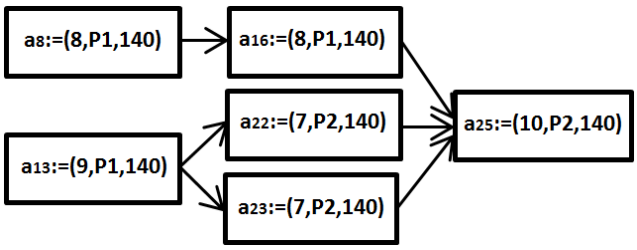

Figure 2: Fragment of motivational example application

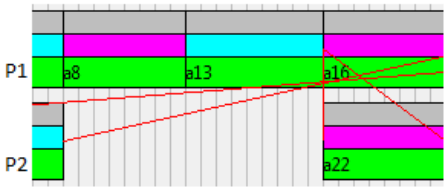

Figure 3: Snapshot of schedule using CALAP

dd. The due-date of a task is the minimum of ($dd_s - e_s$) among all its successors (s) and its own deadline d_t as well as the ($dd(S) - e(S)$) for the set of successors S of the task bound to the same resource. The term $dd(S)$ denotes the maximum among the due-dates of the set of successors on the same resource which is the due-date of the set. The term $e(S)$ denotes the sum of their execution times. This results in a due-date of (*min*(109-(7+7), 140)) = 95 for a_{13}. Now, a_{13} gets preference over a_8 resulting in the schedule a fragment of which is shown in the Gantt chart in Figure 4, with a smaller gap and no deadline misses. This also results in a smaller makespan.

This paper is further organized as follows. Section II presents related work in the domain of multiprocessor scheduling. Section III gives the scheduling context and introduces the required basic scheduling concepts and terminology that are used throughout the paper. It also poses the main problem statement. The complexity of the scheduling problem is analysed in Section IV. Section V introduces the scheduling algorithm. Experimental analysis is detailed in Section VI followed by conclusions in Section VII.

II. Related Work

There is a vast amount of literature available in the scheduling domain. A scheduling problem is defined in terms of the characteristics of the system and the tasks that occur in it. Tasks can be classified as periodic, aperiodic or sporadic as well as preemptive or non-preemptive. Task priorities may be known and they may be either fixed or dynamic. Tasks can be independent or precedence-constrained. Precedence-constrained graphs without cycles are called directed acyclic graphs (DAGs). If tasks have deadlines, then they can be soft real time or hard real time systems based on how strict the deadlines are to the real application. Optimization criteria, such as minimizing makespan, communication latency or throughput, are

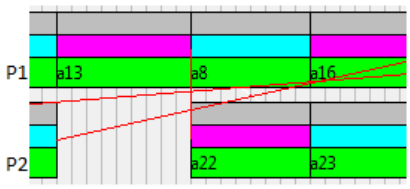

Figure 4: Snapshot of schedule using Due-date

also a means of classifying scheduling problems. Based on platform characteristics, scheduling can be classified into uniprocessor and multiprocessor scheduling. Scheduling problems may be partitioned or global based on whether or not the platform binding of the tasks is known.

The problem being dealt with in this paper is scheduling directed acyclic graphs (DAGs) with deadlines on a multiprocessor platform when the binding is known. There has been significant research done in multiprocessor scheduling for industrial systems presented in the recent survey paper [2]. Among others, it discusses partitioned scheduling without precedence constraints. In [3], the scalability of scheduling large applications on multicore platforms has been addressed and it has been concluded that partitioned EDF is well suited for multiple application types. In our paper, we introduce a partitioned scheduling approach for non-preemptive task graphs that outperforms partitioned EDF. A clustered scheduling algorithm using clusters of processors to compute binding of tasks for soft real-time distributed task systems is presented in [4]. Another recent paper [5] on partitioned scheduling for multicores allows task pre-emption that is not allowed in our problem. Task scheduling for control oriented cyber-physical systems is presented in [6].

The problem of scheduling parallel applications described by directed acyclic graphs (DAGs) on heterogenous grid computing systems is presented in [7]. DAG scheduling with arbitrary length tasks without fixed binding is known to be NP complete [8]. In this paper, we show that the addition of the information on fixed binding does not reduce the complexity of the problem which is still NP-Complete. Significant work has been done on DAG scheduling as presented in [9], which presents a comparison between various DAG scheduling algorithms and their complexities. List scheduling [10] is the most commonly used approach in DAG scheduling, wherein tasks are chosen from a list based on a certain priority and scheduled in that order. There are several heuristics that can be used to efficiently select tasks with list scheduling for better results. Some such heuristic algorithms are the Dominant Sequence Clustering (DSC) [11], Dynamic Level Scheduling (DLS) [12], Modified Critical Path (MCP), Highest Level First (HLF), Longest Path First (LP) and Longest Processing Time First(LPT) [13], [14], [15]. All these heuristics also compute binding of tasks to processing elements. The efficiency of EDF for scheduling sporadic DAG models on multiprocessors platforms with unknown binding is studied in [16]. An algorithm called *FAST* that tries to improve the quality in terms of schedule length of existing scheduling algorithms is presented in [17]. In our paper, we use list scheduling with a new scheduling heuristic that exploits the task binding information known beforehand to make better scheduling decisions.

Work that is close to ours is presented in [18] wherein deadlines are used for the selection heuristic and a deadline modification algorithm is presented. However, this deadline modification algorithm does not take task bindings, if known, into consideration. In our paper, it is shown that this additional information can be used to compute tighter deadlines for tasks. Also, [18] considers unit length tasks with unit communication delays. In our work we have tasks of arbitrary lengths and since the binding is known communication delays are taken into account implicitly in the task graphs. A deadline modification approach for message based scheduling is presented in [19] which takes into account unit communication delays to modify deadlines for scheduling. In [20], classical as-late-as-possible (CALAP) time was presented which comes close to the modified deadlines of this paper but it also does not exploit the binding information. Comparisons are made later in the paper.

III. Problem Definition

A. Context

The computation of a schedule requires information in models based on domain specific languages (DSLs) [21] that are developed by domain experts. The architecture of these models follows the Y-chart approach [22], the layers of which define the application, execution Platform and binding of the application artifacts to platform artifacts as presented in [23]. Essential scheduling information is extracted from these models and model-to-model transformations are performed on them to construct a task dependency graph, consisting of control tasks and their dependencies. Task deadlines are obtained from the latency requirements of the application. Each task is also aware of its resource binding.

B. Preliminaries

We use $\mathbb{R}$ to represent the set of real numbers and $\mathbb{R}^{\geq 0}$ to represent the non-negative real numbers. For a set X, we use X^* to represent the collection of lists with elements from X.

An *application* is a directed acyclic graph (DAG) $G = (T, D)$ with the set of tasks T and the set of task dependencies $D \subseteq T^2$.

The multiprocessor platform that the application is bound to consists of processors called *resources*. R represents the set of resources of the platform.

A *task* $t \in T \in \mathbb{R}^{\geq 0} \times R \times \mathbb{R}^{\geq 0}$ is defined by a tuple $t = (e_t, r_t, d_t) \in \mathbb{R}^{\geq 0} \times R \times \mathbb{R}^{\geq 0}$ where e_t denotes the execution time of the task t, r_t denotes the resource that the task is bound to and d_t denotes the deadline of the task. For

a set A of tasks, execution time $e(A)$ of the set is the sum of the execution times of the tasks and its deadline $d(A)$ is the largest deadline in the set.

D is the collection of task dependencies. *Dependency* $(a, b) \in D$ denotes that task b is allowed to start its execution only after the completion of task a.

A *(static-order) schedule* S is a mapping $S : R \rightarrow T^*$ from the set R of resources to an ordered list of tasks from T. S is a schedule for application $G = (T, D)$ iff (i) every task in T appears once in the ordered list of exactly one of the resources in S, (ii) S respects the task bindings and, (iii) dependencies. During runtime, tasks execute on the resource in the order given by the schedule as soon as the resource is available and the dependencies are satisfied. For task t, we let s_t denote the start time and c_t denote the completion time. The start time of the first task scheduled on a resource is 0. Based on this, the values of s_t and c_t can be computed for each task t given S and G. The completion times are derived from the start times and the task execution times which means that $c_t = s_t + e_t$. A ***feasible schedule*** is a schedule in which all the tasks meet their respective deadlines which means that for all $t \in T, c_t \leq d_t$. During the process of scheduling, we refer to a *partial schedule* S^{prt} which is an intermediate incomplete schedule which does not necessarily contain all the tasks from the application.

The completion time of the last completing task across all resources gives the *makespan* m of the schedule defined by $m = \max_{t \in T} c_t$. The makespan of a resource $r \in R$ is the completion time of the last task on that particular resource given by $m_r = \max_{t \in T, r_t = r} c_t$.

C. Problem Statement

This subsection gives the statement of the exact problem being dealt with in this paper.

Definition 1. *Given an application $G = (T, D)$ with tasks bound to a set R of resources, does there exist a feasible schedule? We call this decision problem* $\mathrm{SP}_{\mathrm{FM}}$*, where* FM *stands for feasible multiprocessor scheduling.*

The execution times of tasks are fixed values extracted from measurements on the machine. Since the task bindings are known a priori, communication time is considered implicitly in the execution times of existing tasks in the task graphs and hence need not be considered separately during scheduling.

IV. Complexity Analysis

It is well known that Multiprocessor scheduling of DAGs under latency constraints with unknown binding is NP-complete [8]. This raises the question whether the additional binding information allows a more efficient solution. In [24], it is shown that determining the existence of schedules with a makespan of at most four for a collection of fork graphs with unit execution tasks, with known binding on m processors and without communication delays, is NP Complete. A fork graph is an outtree of height one that consist of a task as its source and the children of the source as its sinks. The DAGs in $\mathrm{SP}_{\mathrm{FM}}$ are a generalization of a collection of fork graphs. Also, our DAGs consist of tasks with arbitrary execution times as opposed to unit execution tasks. The required makespan of at most four can be trivially converted to a task deadline of four for all tasks. All this combined with the fact that communication delays are implicit within the tasks in $\mathrm{SP}_{\mathrm{FM}}$, imply that $\mathrm{SP}_{\mathrm{FM}}$ is a generalization of the problem considered in [24]. Hence, $\mathrm{SP}_{\mathrm{FM}}$ is NP-hard.

The NP-Completeness of SP_{FM} can be proven by showing that it belongs to the class NP. A decision problem belongs to class NP if the solution to the problem is verifiable in polynomial time. Consider an instance of $\mathrm{SP}_{\mathrm{FM}}$ and a schedule S, we need to verify if S is a feasible solution. Given the execution times of tasks, the data dependencies and the ordering of tasks on their respective resources, the completion time of tasks can be computed using a function that is linear in the number of data dependencies in the task graph. Since a graph with m tasks can have $(m \times m)$ data dependencies, this function is quadratic in the number of tasks. This means that the completion times can be computed in polynomial time. The check for feasibility is efficient, since it involves a simple check to ensure that the completion time does not exceed the deadline for each task. This shows that given a schedule, it can be verified in polynomial time. Thus, $\mathrm{SP}_{\mathrm{FM}}$ belongs to class NP and is NP-complete leading to Theorem 1.

Theorem 1. *SP_{FM} is NP-complete.*

The following section elaborates on the scheduling algorithm.

V. Scheduling Algorithm

In this section we describe our scheduling algorithm and its heuristic to arrive at a feasible solution as quickly as possible.

A. List Scheduling and Static Order Schedules

We perform scheduling on a per task basis using list scheduling with a new heuristic. The scheduler keeps track of the set of enabled tasks and a partial schedule. Each time, a task is chosen from a set of enabled tasks using the heuristic and is consequently scheduled on its resource. Its start time s_t is computed as the maximum of the completion times of its predecessor tasks and the earliest time that the task can be scheduled on the resource. This may lead to gaps in the schedule when a task cannot be scheduled immediately after the last scheduled task on a resource due to task dependencies. The scheduler also tries to fit tasks into existing gaps in the partial schedule. So, the earliest time on the resource also considers the earliest time among all the gaps in the partial schedule that are large enough to hold

the task. Once the task is scheduled, the partial schedule is updated to include information of this scheduled task. When a task completely fits into a gap, the gap is removed from the updated schedule. If the gap is bigger than the size of the task then the task is scheduled in the gap and the gap is updated to a smaller sized gap in the schedule. Using the updated schedule, the new set of enabled tasks is computed from the data dependencies in the task graph and the process is repeated until all tasks are scheduled or the schedule is found to be infeasible and the scheduler returns with a message indicating this. This process is explained later in Section V-C.

In order to improve the odds of arriving at a feasible schedule without backtracking, there is a need to smartly select which task is to be scheduled from the set of enabled tasks. The heuristic we use for this purpose is based on a derived property of a task called *due-date*, introduced next.

B. Due-dates

A due-date $dd \in \mathbb{R}$ of a task is an upper bound on the completion time of the task which, if missed, renders the scheduling problem infeasible. If dd is a due-date of a task $t \in T$, then any $dd' > dd$ is therefore also a due-date. Also, dd is a due-date of a set A of tasks, if dd is a due-date for all $t \in A$. If specific due-dates (dd_t) are known for all $t \in A$, then the due-date of A is $dd(A) = \max_{t \in A} dd_t$. Due-dates are used to determine as soon as possible during scheduling whether a particular branch in the search space is infeasible or not hence allowing us to prune the search space by omitting the infeasible branches. They are also used in the heuristic to choose a task from the set of enabled tasks and steer the scheduling process. A task with a smaller due-date is more urgent, which leads to the scheduling heuristic called *Earliest Due-Date First* wherein a task with a smaller due-date is chosen for scheduling sooner than a task with a larger due-date. This is the same as the Earliest Due-Date heuristic (EDD) introduced by Jackson in 1955. The challenge is to find the tightest (smallest) possible due-date for a task since this enables us to detect sooner during scheduling whether a particular branch is infeasible or not. It also enables us to better determine which is the most urgent task in the set of enabled tasks and steer the scheduler in the right direction. However, finding the tightest possible due-date is as hard as solving the scheduling problem. The corresponding decision problem is NP-Complete as explained in Theorem 2

Theorem 2. *Given a task t and an arbitrary number k, deciding whether its tightest due-date dd_t is greater than or equal to k is NP-Complete.*

Proof: This theorem is proved in two steps. The first step shows that the decision problem is NP-Hard, followed by a proof of its NP-Completeness. We prove the NP-Hardness by showing a reduction from SP_{FM} which is already proven to be in NP-Complete in Section IV. We show that any instance of SP_{FM} can be reduced to an instance of this problem where $k = 0$. To illustrate this, consider any instance of SP_{FM} with a task graph T. We make t the initial task to T such that t has outgoing dependencies to all the source tasks in T. If there does not exist a feasible schedule to T, it implies that $dd_t < 0$. Alternatively, if there exists a feasible schedule to T, $dd_t \geq 0$. Hence, solving SP_{FM} gives the solution to an instance of this problem where $k = 0$, implying that it is at-least as hard as SP_{FM}. Hence, it is NP-Hard.

Given the tightest due-date of t and an arbitrary k, it is trivially known whether $dd_t \geq k$, implying that the solution to the problem is verifiable in polynomial time. Hence, the problem belongs to the class NP and is NP-Complete. ■

Our purpose is to compute due-dates of tasks prior to scheduling and use the earliest due-date first heuristic to perform scheduling efficiently in such a manner that feasible schedules can be arrived at without the need for backtracking, hence saving time. So to simplify the due-date computation we present a means to extract a due-date from the immediate successors of a task in the task graph, since this information is easily known and does not require the exact schedule information. Computing due-dates in a particular order by traversing the task graph backwards ensures that even the future successors of a task are accounted for in its due-date computation. Along with looking at the successors, we also look at the resources that the successors are bound to. We use this information to compute tighter due-dates as explained later in this section. The due-date so computed is the due-date we use in the earliest due-date first heuristic in this paper.

As mentioned earlier, computation of due-dates of tasks is done starting with the sink nodes, i.e. nodes with no outgoing dependencies, in the task graph and traversing backwards in topological order of dependencies. The due-date of a task depends on its own deadline and the due-dates of its immediate successors. In the absence of any successors, the due-date of a task is simply its deadline. As mentioned in Section II, a deadline modification approach is presented in [18] for the case of dynamic binding. In [20], classical ALAP times are computed by taking the minimum of the $dd_t - e_t$ for all successor tasks. However, using the binding information, successors of a task can be partitioned into groups deployed on each resource. We exploit the fact that all successor tasks on one resource must be completed before the largest of their due-dates. This is because tasks with a due-date smaller than the largest must be completed before their respective due-dates, and the tasks with the largest due-date must be completed before that largest due-date. The due date computation of a task first computes the due-date per resource, which is the due-date of the task considering only the successors bound to that particular resource, and then takes the minimum of the due-dates, so computed, over all resources. These two steps are explained below:

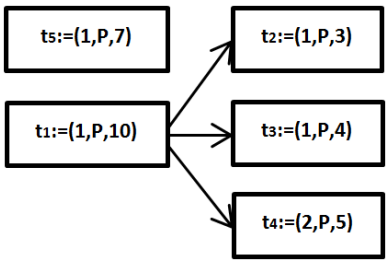

Figure 5: Due-date computation

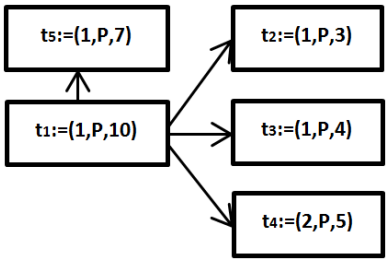

Figure 6: Non-monotonicity of due-date computation due to addition of a dependency

Due-date per resource: When a set of successors of a task is bound to the same resource, all these successors must be completed sequentially before their largest due-date. For example, in Figure 5, task t_1 has 3 successors. These tasks have no successors implying that their deadlines are their due-dates. Task t_2 has to be completed before time 3, task t_3 before time 4 and task t_4 before time 5. This means that all three successors must be completed before their largest due-date, 5. Task t_1 has to be scheduled before all of them and they are executed sequentially on the resource. Subtracting the sum of the execution time of all successor tasks from the largest due-date gives a due-date for t_1. Hence, 5-(1+1+2) = 1 is a due-date. By doing this, we are exploiting the additional binding information of the successors that is known to us. We also get the due-dates from looking at the $dd_t - e_t$ of the single successors: 5-2 = 3, 4-1 = 3 and 3-1 = 2. We use the minimum among all these due-dates as the due-date of the task which might be an equal or tighter value than just taking the minimum of the $dd_t - e_t$ of the individual successors. This gives us a due-date of *min*(1, 3, 3, 2) = 1 in Figure 5. Note that any subset of successors can be used to compute a due-date.

An even tighter due-date can be achieved by calculating the due-dates in this manner for all possible subsets of the successor tasks. Doing this ensures that adding an extra dependency from a task to another task in the task graph does not cause its due-date to relax. This is not the case using just the above explained computation as illustrated using the example in Figure 6, where a new dependency is added from task t_1 to another task t_5 in the task graph which has a due-date of 7 time units and an execution time of 1 time unit. The resulting due-date of t_1 is *min*(7-(1+1+1+2), 7-1, 5-2, 4-1, 3-1) = 2 time units. However, we know from the earlier computation that there exists a tighter due-date which can be computed if we exclude the newly added task from the set of successors. Hence, it is necessary that this due-date computation is performed by considering all subsets of the successor tasks and taking the minimum. This computed value will be unique since we have taken into account all possible ways the successor tasks can be scheduled after the current task. In Figure 6, one of the subsets $\{t_2,t_3,t_4\}$ gives us the tighter due-date of 1 time unit. Computing due-dates using all successor subsets needs us to compute due-dates for all the 2^n possible subsets for n successor tasks and take the minimum among them. However, among these 2^n subsets several subsets are not necessary to obtain the due-date as elaborated in Theorem 3.

Theorem 3. *The due-date of a task with n successors computed using all 2^n successor subsets is the same as the due-date of a task computed using n subsets chosen incrementally by adding one successor at a time from the set of successors in descending order of due-dates.*

Proof: Consider the successor with largest due-date. In the example on Figure 6 it is the task t_5. Keeping this task fixed in the subsets, there are 2^{n-1} possible subsets which include this task. Among these only one subset containing all successor tasks is relevant and gives the tightest due-date owing to the fact that all successors in the subset need to be completed before the due-date of t_5. As a result we can consider this one subset instead of considering the 2^{n-1} subsets including the task t_5. This same rule applies to even the other successors of the task. For every successor we only consider one subset which contains itself and all tasks with a due-date smaller or equal to its own. We end up with exactly n subsets, one for each successor. The resultant due-date which is the minimum among the due-dates computing using these n subsets, is the same as the one obtained by taking the minimum of all subset due-dates. ■

Considering subsets in the incremental manner explained above in the naive implementation for one particular task is still quadratic in the number of successors of the task since the due-date computation is repeated for each successor being added to the subset. An alternative linear approach for computation of the due-date of a task t on a particular resource r computes the same result without needing to perform the repeated computation for all the n successor subsets. It starts with an infinite due-date and sorts the successors in descending order of due-dates. Starting with the first successor in the sorted list, it assigns the $dd_t - e_t$ of this successor as the due-date of the task. Thereafter it keeps deducting the execution time of subsequent successors in the sorted list from this value as long as the result is equal to or smaller than considering the $dd_t - e_t$ of the successor being considered. In case it is larger, the value computed till then is discarded and the new due-date of the task is $dd_t - e_t$ of

Algorithm 1: computeDueDatePerResource()

Input : $G := (T, D), R, t, r$
Output: dd_t

1 $dd_t := \infty$;
2 **for** $t' \in succ_r^{ordered}(t)$ **do**
3 **if** $dd_{t'} \leq dd_t$ **then**
4 $dd_t := dd_{t'} - e_{t'}$;
5 **end**
6 **else**
7 $dd_t := dd_t - e_{t'}$;
8 **end**
9 **end**
10 **return** *min*(dd_t, d_t)

this successor. The execution times of subsequent successors in the list are then extracted from this value. Proposition 1 shows that this is equivalent to considering only those smaller subsets that can give a tighter due-date for the task and ignore considering all subsets in between since they do not contribute to tightening the computed due-date value in any manner.

Proposition 1. *For a task with sets A and B of successors such that $A \subset B$, the due-date computed using A is tighter only if $dd(A) < (dd(B) - e(B - A))$.*

Proof: Since $A \subset B$, B has all the successors in A plus at-least one more. The due-date computed for subset A is $dd(A) - e(A)$. The due-date of subset B is $dd(B) - e(B)$ which in turn is $dd(B) - e(B - A) - e(A)$ since $A \subset B$. As such, considering subset A will yield a tighter due-date only if $dd(A) < dd(B) - e(B - A)$. ■

The minimum of the resultant value with the original deadline of the task gives its due-date. This alternative due-date computation for a particular task is linear in the number of successors to the task. Consequently, the due-date computation for the entire task graph is quadratic in the number of tasks. Let $succ_r^{ordered}(t)$ represent the list of immediate successors of a task t on its resource r, in descending order of due-dates. Algorithm 1 shows this approach.

Due-date across resources: The minimum over the due-dates computed per resource is a due-date of the task.

C. List Scheduling with Earliest Due-date First Heuristic

Algorithm 2 formalizes the scheduling process. Let all predecessors of a task t be denoted by $\text{pred}(t) := \{t' \in T \mid (t', t) \in D\}$ and all successors of t be denoted by $\text{succ}(t) := \{t' \in T \mid (t, t') \in D\}$. A gap g in the schedule, is defined by the tuple $g = (s_g, c_g) \in \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$, where s_g represents the start time of the gap and the c_g represents the closing time of the gap. Let G_r represent the set of gaps left on a resource $r \in R$ during scheduling. The algorithm

Algorithm 2: Scheduler()

Input : $G := (T, D), R$
Output: S

1 $\forall r \in R, S_r^{prt} := NULL$;
2 $computeDueDates(G, R)$;
3 $ST := \phi$;
4 **while** $|ST| \neq |T|$ **do**
5 $ET := \{t \in T \mid pred(t) \subseteq ST\}$;
6 $t := \arg\min_{t \in ET} dd_t$;
7 $c_{lastPred_t} := \max_{t' \in pred(t)} c_{t'}$;
8 $g_{chosen} := \arg\min_{g \in G(r_t)} \{s_g \mid \max(c_{lastPred_t}, s_g) + e_t \leq c_g\}$;
9 **if** $g_{chosen} \neq NULL$ **then**
10 $s_t := \max(c_{lastPred_t}, s_{g_{chosen}})$;
11 **end**
12 **else**
13 c_{last_r} := Completion time of last task on r;
14 $s_t := \max(c_{lastPred_t}, c_{last_r})$;
15 **end**
16 **if** $c_t > dd_t$ **then**
17 **return** *'deadline miss'*;
18 **end**
19 **else**
20 $S_{r_t}^{prt} := append(S_{r_t}^{prt}, (e_t, r_t, d_t))$;
21 $ST := ST \cup \{t\}$;
22 $ET := \{ET \backslash \{t\}\} \cup \{t' \in succ(t) \mid pred(t') \subseteq ST\}$;
23 **end**
24 **end**
25 **return** S;

first computes the due-dates of all tasks in the task graph in line 2. It then maintains a set of enabled tasks (ET) and another set of scheduled tasks (ST) which is initially empty. It schedules enabled tasks in a loop until all tasks in the task graph are added into the set of scheduled tasks. In line 5, the current set of enabled tasks are extracted as the ones whose predecessors have been scheduled. In line 6, the task with the earliest due-date t is chosen from the set of enabled tasks. The completion time of the last completing predecessor ($lastPred_t$) of t is extracted in line 7. If there are any gaps in the schedule, the chosen gap g_{chosen} is the earliest gap that is large enough to fit t after the completion time of its last completing predecessor as given in line 8. In lines 9-11, if there is such a gap, the task is scheduled in it starting from either the beginning of the gap or somewhere in between if any of its predecessors completes later. In lines 12-15, if there are no gaps, arg min of the empty set returns *NULL* and the task is scheduled at the end of the partial schedule on its resource either immediately after

the last task scheduled on r ($last_r$) or at any later time depending on the completion times of its predecessors. In lines 16-18, if the completion time of the task exceeds its due-date, either the current or some successor task will miss its deadline and the resultant schedule will be infeasible. At this point, it could be decided to backtrack and chose tasks differently to arrive at alternative schedules. The possibility of incorporating backtracking has not been elaborated here. If the task does not miss its due-date, it is added to the partial schedule on its resource as well as to the set of scheduled tasks in line 20 and 21. Following this in line 22, the new set of enabled tasks are obtained by removing this task from the current set and adding all other tasks which are now enabled due to this task into the set. The whole process then repeats.

VI. Experimental Evaluation

A. Industrial test cases

As mentioned in the introduction, task graphs originating from the wafer scanner control domain have been scheduled using the scheduler introduced in this paper. In these task graphs, latency requirements of the control blocks have been specified as deadlines. The binding of a block to its processor resource is given along with its execution times on the resource. Using the information in the control application, schedules are computed for each processor resource in the platform. For the purpose of our study, three separate control applications from the wafer scanner systems have been chosen as described below. The control tasks range from simple operations like addition to complex operations like filtering, clipping and matrix multiplications that are performed by larger control tasks.

NXT Motion: NXT is a version of the ASML wafer scanners called TWINSCAN. These can process two wafers at a time wherein one wafer is being measured for its accurate positioning and orientation and a different wafer is being exposed with an electronic circuit. It comprises of several components that move synchronously to transfer the wafers within the system. A large number of sensors and actuators are required to perform the measurements and the movements accurately to ensure perfect wafer movement and positioning. The complete NXT motion control application contains 4301 tasks and 4095 dependencies. These tasks are scheduled on a platform consisting of 11 single core processors and run with a frequency of 10kHz. The execution time of the tasks range from a minimum of $1 \cdot 10^{-8}s$ to $2.25 \cdot 10^{-6}s$. The tasks are assigned 72% of the processor budget and the remaining is kept for background processing, which gives them a deadline of $7.2 \cdot 10^{-5}s$. Apart from this, the task graphs also contain several critical actuator tasks that are assigned tighter deadlines. They are critical because the time that data is received by actuators determines the delay of the particular component and is crucial to the throughput of the machine. In this case, actuators are assigned a deadline of 30% of the processor budget giving them a deadline of $3 \cdot 10^{-5}s$. The number of incoming and outgoing dependencies (degree) of the tasks range from 0 to 22 dependencies. Apart from this, there is also a specialized NXT Motion application which assigns short deadlines to a special set of actuator tasks involved in a critical movement operation in the machine. These tasks are of higher priority and are assigned 26% of the processor budget giving them a deadline of $2.6 \cdot 10^{-5}s$.

Flexray: The lithography process to expose wafers requires light beams from a light source to be passed through the transparent and opaque regions on a quartz plate containing the images of the circuits, called a reticle, onto the wafer. The illuminator forms a key part of this lithographic optical system. It conditions the light from the source, and causes the light beam to take on a prescribed shape, known as the pupil shape, before it goes through the reticle. This component, called the Flexray, is involved with imaging and includes 4096 mirrors working in parallel to transmit a light beam accurately onto a wafer. As such it contains an extremely large number of parallel control tasks which manipulate each mirror independently in 6 degrees of freedom to adjust the pupil shape. This is the largest model that the schedulers have been tested on, containing as many as 14,908 tasks and 26,189 dependencies. These tasks are scheduled on a platform consisting of 14 single core processors which run on a sample frequency of 238 Hz. The execution time of the tasks range from a minimum of $1 \cdot 10^{-9}s$ to as much as $8.31 \cdot 10^{-5}s$. The tasks are assigned 100% of the processor budget, since it a very large application, which gives them a deadline of $4.201 \cdot 10^{-3}s$. The degrees of the tasks range from 0 to 131 dependencies.

Projection optics box (POB): The latest EUV/NXE wafer scanner systems make use of UV light. The Projection Optics Box (POB) is the enclosure containing the optical components (mirrors) between the reticle and the wafer stage that holds the wafers to be exposed. Since air absorbs the EUV light, the POB is kept under vacuum to ensure that there is no loss of intensity in the light falling on the wafer. The POB models contain 1878 tasks and 1540 dependencies. These tasks are scheduled on a platform consisting of 6 single core processors which run on a sample frequency of 10kHz. The execution time of the tasks range from a minimum of $3.42 \cdot 10^{-8}s$ to $1.36 \cdot 10^{-6}s$. The tasks are assigned 70% of the processor budget, with the rest being kept for background processing, which gives them a deadline of $7 \cdot 10^{-5}s$. Actuator tasks are assigned a deadline of 22% of the processor budget. The degrees of the tasks range from 0 to 13 dependencies.

Table I gives the due-date computation and scheduling times obtained after running the scheduler on each of these control applications. To draw comparisons, we ran the above test cases with a list scheduler using the earliest deadline first (EDF) heuristic and we observed that the scheduler fails for all four cases. We also performed a comparison of

Table I: Measured timings for due-date computation and scheduling

Application	Due-date Computation (s)	Scheduling (s)
Motion	0.05	1.15
Specialized Motion	0.05	1.14
Flexray	0.18	3.64
POB	0.03	0.81

Table II: Comparison between EDDF, EDF and ECF using industrial test cases

Application	EDDF	EDF	Makespan difference	# Task deadline misses	ECF	Makespan difference	# Task deadline misses
Motion	Feasible	Infeasible	$9.57 \cdot 10^{-5}$	382	Feasible	0	0
Flexray	Feasible	Infeasible	$4.99 \cdot 10^{-4}$	563	Feasible	0	0
POB	Feasible	Infeasible	0	324	Feasible	0	0
Specialized Motion	Feasible	Infeasible	$9.57 \cdot 10^{-5}$	382	Infeasible	0	44

Table III: Comparison of feasibility results of EDDF and ECF on synthetic test cases

Outcome	ECF Feasible	ECF Infeasible
EDDF Feasible	254	336
EDDF Infeasible	263	147

Table IV: Comparison of makespan results of EDDF and ECF on synthetic test cases

$\textbf{EDDF}_{Makespan} < \textbf{ECF}_{Makespan}$	$\textbf{ECF}_{Makespan} < \textbf{EDDF}_{Makespan}$	$\textbf{EDDF}_{Makespan} = \textbf{ECF}_{Makespan}$
403	344	253

the earliest due-date first (EDDF) heuristic with the earliest CALAP first (ECF) heuristic. The two heuristics produce similar schedules in most cases. We see a difference in the specialized NXT-Motion case. This is a specialized and highly constrained application for which the list scheduler with the earliest due-date first heuristic produces a feasible schedule in the first shot as opposed to the list schedulers with the earliest CALAP first heuristic that fails to produce a feasible schedule in the first shot. The total makespans of the two schedules are the same for all applications. The results are summarized in Table II.

B. Comparison of EDDF and ECF: Synthetic test cases

To evaluate the statistical significance of improvements obtained by our approach over ECF, the schedulers have been run on 1000 directed acyclic graphs that are generated using the random graph generator tool of SDF3 [25]. These task graphs with 4500 tasks each are bound to 2 to 5 processors using a binding approach that binds entire branches containing subgraphs to the same resource as much as possible. This is similar to the binding approach used in ASML which binds groups of interconnected control tasks on the same resource in order to minimize communication delays across resources, as opposed to task binding where maximum parallelism is utilized. The execution times of the tasks randomly vary from $1 \cdot 10^{-2}s$ to $30s$ with an average of $2s$ in accordance with the ranges scaled up for the NXT-Motion application. The tasks are assigned deadlines which is twice the average load on the resources. Between 200-250 tasks are randomly chosen to be critical in accordance to the number of critical actuator tasks in the NXT-Motion application. These tasks are assigned deadlines equivalent to the average load on the resources. The degree of the tasks ranges from 1 to 10 with an average of 5. EDDF produced 590 feasible schedules, whereas ECF produced 517 feasible schedules. There were 336 test cases where EDDF produced feasible schedules and ECF produced infeasible schedules. On the other hand, there were 263 test cases where ECF produced feasible schedules and EDDF produced infeasible schedules. There were 254 test cases where both EDDF and ECF produced feasible schedules, and 147 cases where both produced infeasible schedules. These results are summarized in Table III. To ensure that the positive result for EDDF is not a coincidence, we performed the McNemar test on these results and obtained a very low p-value of 0.001 which indicates that the probability that this positive outcome of EDDF is coincidence is only 0.1%. Apart from this we also compared the makespans obtained using the two approaches. EDDF and ECF produced identical makespans in 253 of the 1000 test cases. EDDF produced a better (smaller) makespan than ECF in 403 cases and ECF produced a better makespan in 344 test cases. These results are summarized in Table IV. We performed the paired student t-test on the values of the makespan obtained using EDDF and ECF. A very low p-value of $7.32 \cdot 10^{-6}$ was obtained, which is also in favour of our claim that the positive outcome of EDDF is not a coincidence.

VII. Conclusions and Discussions

The multiprocessor scheduler presented in this paper has been shown to compute schedules of very large task graphs within hard real-time constraints. The measured timings

showed the effectiveness of the approach for very large task graphs and hence this approach will be incorporated in the next versions of ASML's machines.

The tasks in the industrial test cases scheduled by the scheduler have fixed measured execution durations. Most of the time, tasks execute with execution times around these durations. However, the platform is composed of general purpose processors (GPP) that are designed to minimize the average case execution time. There are chances that tasks execute with their worst case durations due to memory contentions, cache misses, branch predictions, etc. Tasks are assigned deadlines which are derived from assigning around 70% of the processor budget to sample processing; the remaining budget used for background processing. As a result, these machines can tolerate cases where a task executes in its worst case and misses its deadline. Moreover, scheduling for the worst case produces schedules that are too pessimistic for the average case since the likelihood of all tasks executing in their worst case is extremely low. However, given the above observations it will be an interesting future improvement to the scheduler if it can use the remaining scheduling freedom to produce schedules that are more robust to these task variations.

REFERENCES

[1] "ASML," http://www.asml.com, accessed: 21/02/2013.

[2] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[3] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proceedings of the 2008 Real-Time Systems Symposium*, ser. RTSS '08.

[4] C. Liu and J. Anderson, "Supporting graph-based real-time applications in distributed systems," in *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

[5] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, July 2009, pp. 239–248.

[6] F. Zhang, K. Szwaykowska, W. Wolf, and V. Mooney, "Task scheduling for control oriented requirements for cyber-physical systems," in *Real-Time Systems Symposium*, 2008.

[7] A. Forti, "Dag scheduling in grid computing systems," Ph.D. dissertation, 2006.

[8] C. Papadimitriou and M. Yannakakis, "Scheduling interval-ordered tasks," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 405–409, 1979.

[9] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[10] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961.

[11] T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, Sep 1994.

[12] G. Sih and E. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, Feb 1993.

[13] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974.

[14] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, vol. 5, no. 1, pp. 23–32, Jan 1988.

[15] M.-Y. Wu and D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, July 1990.

[16] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[17] M.-Y. Wu, W. Shu, and J. Gu, "Efficient local search for DAG scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2001, pp. 617–627, 2001.

[18] J. H. Verriet, "Scheduling with communication for multiprocessor computation," Utrecht University, Tech. Rep., 1998.

[19] T. F. Abdelzaher and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 11, pp. 1179–1191, 1999.

[20] A. H. Timmer, "From design space exploration to code generation," Ph.D. dissertation, 1996.

[21] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, 2008.

[22] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*. IEEE Computer Society, 1997.

[23] R. Schiffelers, W. Alberts, and J. Voeten, "Model based design, analysis and synthesis of servo controllers for lithoscanners." MPM, 2012.

[24] J. Verriet, "The complexity of scheduling typed task systems with and without communication delays," Utrecht University, Tech. Rep., 1998.

[25] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, 2006. [Online]. Available: http://www.es.ele.tue.nl/sdf3