MASTER

Automated multi-core deployment for high-end servo control

Ok, B.

Award date: 2015

Link to publication
5T746 Master Thesis ES
Automated Multi-Core Deployment for High-End Servo Control

Beril Ok
January, 2015

TU/e Supervisor:
dr. ir.Jeroen Voeten

ASML Supervisor:
dr. ir.Ramon Schiffelers

Tutor:
ir.Shreya Adyanthaya

Eindhoven University of Technology, 2015.
# Contents

1 Introduction .............................................. 5  
1.1 CARM ............................................. 5  
1.2 Problem Domain ................................. 6  
1.3 Problem Definition ......................... 8  
1.4 Research Hypothesis .................... 8  
1.5 Related Work ............................... 10  

2 Scheduler ............................................ 13  
2.1 Preliminaries ............................. 13  
2.2 Inter-worker Dependencies .......... 14  
2.3 Intra-worker Dependencies .......... 15  
2.4 Scheduling .............................. 17  
   2.4.1 Single-PU Scheduling .......... 17  
   2.4.2 Multi-PU Scheduling .......... 17  

3 Manual Deployment ......................... 20  
3.1 Manual Deployment Strategy .......... 20  
3.2 Results .................................. 23  
   3.2.1 Case 1: POB mirror network .... 23  
   3.2.2 Case 2: Stages Stack .......... 25  
3.3 Conclusion ............................ 26  

4 Automatic Deployment ....................... 28  
4.1 Automatic Deployment Strategy ...... 28  
4.2 Results ................................ 31  
4.3 Drawbacks ................................ 33  
4.4 Conclusion ............................ 35  

5 Algorithmic Detection of the Application Structure and Automatic Deployment .......................................... 36  
5.1 Implementation .......................... 36
Abstract

This report describes the results of the Master Graduation Project in the Embedded System Program of Technical University of Eindhoven which is carried out within the CARM 2G team at ASML. The report comprises of the problem statement, a literature study for the background knowledge and possible solutions with some experiments for the stated problem. The objective of this project is to improve the latency of the servo controllers used in lithoscanners of ASML. For this purpose a number of possible improvements of the multi-core deployment strategy of the scheduler applied within ASML are investigated. The current scheduler is designed such that deployment decisions are made without having global application knowledge. It is shown that the performance of the scheduler can be improved by exploiting this global application structure. Three approaches are investigated. In the first two approaches the structure present in the application is used, first by giving manual deployment hints to the scheduler and secondly by computing deployments algorithmically. From these approaches we conclude that exploiting global application structure indeed improves timing performance. Disadvantage of these approaches however is that the given application structure is not optimal to exploit concurrent execution. Therefore in the third approach the structure targeting application parallelization is extracted automatically from the application graphs. For this, the Dominant Sequence Clustering (DSC) approach is selected.
Chapter 1

Introduction

ASML is the world’s leading provider of lithography systems for the semiconductor industry. Wafer scanners developed at ASML consist of many servo control systems having tight hard real time requirements. In order to meet the required machine performance, applications have to run at high rates and satisfy stringent latency requirements between sensing and actuation. A model driven design space exploration approach is used in the optimization of the timing performance of these systems. This approach gives the opportunity to explore in an early stage possible design solutions in a systematic way, so that a disappointing system performance will be avoided [16].

1.1 CARM

Within ASML, an execution platform that conforms to the so-called Control Architecture Reference Model (CARM) has been developed which supports the design and implementation of the servo control systems within wafer scanners [12]. The core of CARM is based on a set of domain specific languages (DSLs) which describe the control applications, multi-core execution platform and the mapping of the applications to the execution platform as shown in Figure 1.1. The mapping includes the scheduling of the control tasks and deployment of the tasks to the resources. Platforms consist of workers containing one or more processing units (depending on whether single-core or multi-core platform is used). Processing units are the entities that carry out the computations defined by tasks in the application. The application mapping requires course grain deployment for the mapping of servo groups onto workers and a fine grained deployment for the mapping of control tasks to the processing units.
1.2 Problem Domain

Applications are Synchronous Homogeneous Data Flow graphs in which the nodes of the graph correspond to the tasks and the edges in the graph correspond to control and data dependencies between the tasks. Applications are mapped and scheduled on a platform in several steps which are shown in Figure 1.2.

In the first step, the application graph is refined with additional tasks.
and dependencies. These additional tasks are based on the control modes of operation which are obtained after the start-up of the machine. This refined graph is then transformed into a Directed Acyclic Graph (DAG) which is used as input for the scheduler. In addition, the scheduler takes end-to-end timing requirement and tries to compute a deployment and schedule that satisfies these requirements.

Figure 1.3 shows the groups of servo applications (SGs) that are deployed onto specific groups of processing units called Workers. The servo group to worker deployment is fixed and offline. This is called coarse grained deployment and is visualized by the solid arrow in Figure 1.3. SGs have block groups (BGs) and BGs consist of tasks. Dashed arrows in Figure 1.3 show that the fine grained deployment maps the tasks within the BGs onto the processing units assigned by coarse grain deployment. The fine grained deployment is computed by the scheduler.

During the initial development of the scheduler, the synchronization timing between processing units within a worker was assumed to be negligible. However, later measurements on realizations showed that this assumption caused large sample times and incorrect predictions of the make-span. Hence, synchronization between processing units needed to be taken into account.
To ensure data flow semantics, a guard and update based mechanism was developed that adds guard and update tasks to perform the synchronization. However, due to many pu2pu dependencies, several unnecessary guard and update tasks were added in the schedule resulting in a big make-span. These are minimized using a transitive reduction algorithm with returns only the required guards and update corresponding to non-redundant dependencies. [4].

The current scheduler makes the decision for small tasks to be deployed onto the same processing units and large tasks to be executed in parallel. These decisions are made locally lacking the structural knowledge of the whole application. Consequently, the resultant schedules still contain a large amount of communication overhead between the processing units and can still be improved.

1.3 Problem Definition

In control systems, IO delay and sampling frequency are closely coupled parameters which affect the performance of the system. The sampling frequency is the rate at which inputs from sensors are read and outputs to actuators are written. The IO delay refers to the time between sensing and actuating. A schedule typically starts with tasks that read sensor data. Then a collection of tasks follow to compute the output, followed by a task that produces output data. Then a number of tasks follow that prepare for the next sample.

The objective of this project is to reduce the IO delay and to improve the sample frequency of the servo control application at ASML. The means to achieve this is to reduce the number of dependencies that cross different processing units within a worker.

1.4 Research Hypothesis

For every task the scheduler determines the processing unit it will be deployed on. Initially the scheduler was developed by assuming zero synchronization overhead between processing units. This implied basically that the scheduler would pick any processing unit on which a task would finish earliest.

Later when it became clear that processing unit synchronization takes a considerable amount of time, the synchronization penalty was taken into account. For two tasks with a dependency in between, this implies that if one task is deployed on a processing unit, the other one will only be deployed on
another processing unit if the synchronization penalty does not exceed the gain obtained by exploiting concurrency. In practice this means that tasks with small execution times are often mapped on the same processing unit, while tasks with large execution times are run in parallel.

The deployment decisions are made by the scheduler without exploiting the structure of the application. However, our motivating example below suggests that sample rates and IO delay can still be improved by explicitly taking this structural knowledge into account. This leads to our research hypothesis:

"The sample frequencies and IO delay can be improved through better deployment strategies that exploit global structural application knowledge by reducing the number of required synchronizations between processing units."

A motivating application example is given in Figure 1.4. It suggests that the synchronization timing can be reduced by a better deployment strategy. As explained before, the edges in the application graph show the dependencies between the tasks. If dependent tasks are deployed onto different processing units, communication and synchronization timing result in a timing penalty.

Assume that all the tasks have an execution time equal to one unit of time and that a two-PU worker is used. The dependencies between PUs (the arrows between the tasks) are responsible for additional latency in the system. The resultant schedule obtained by the current scheduler is shown in Figure 1.5. The current algorithm deploys the tasks such that the resultant schedule has a make-span of 7 units of time. However, if we look at the hand-made deployment proposed in Figure 1.6, taking into account the structure
of the application results in a make-span of 6 units with a lower number of dependencies between processing units. In Figure 1.5, scheduler’s decision to deploy task 3 onto PU2 results in synchronization timing penalty between tasks 1 and 3 and tasks 3 and 4. By deploying task 3 into PU1 with the tasks that it has most dependency results in fever communication overhead in the overall schedule. This can be achieved by the global application structure knowledge. This small example suggests that the schedule can be improved by exploiting the global application knowledge and feeding it to the scheduler. It is expected that deploying the most dependent tasks into same processing units which is revealed by the global knowledge will result in a better make-span.

![Figure 1.5: Schedule by the current scheduler](image)

![Figure 1.6: Proposed Schedule](image)

### 1.5 Related Work

There is a vast amount of literature study in the scheduling and the deployment domain. The scheduling problem is defined to be an NP-complete. The optimization criteriasuch as minimizing make-span and IO delay is a part of the scheduling problem so as the deployment.
The deployment strategies which use the existing structure proposed in this report are inspired by the domain analysis for the ASML machines. First of all, in the manual deployment, it is proposed to exploit the inherent parallelism in the wafer positioning module used in the lithoscanners of ASML since it contains the 6 degree of freedom blocks. Secondly, it is shown that the applications have hierarchical structure defined by the domain experts concerning the control engineering. It is proposed that the parallelism can be exploited in this hierarchical structure. The assumption is that the dependency between the tasks increases towards the lower layer of the hierarchy.

In order to further exploit the parallelism in the applications, the clustering algorithm is proposed. There are similar algorithms in the literature that all aims to exploit the parallelism in the applications. Edge zeroing technique is proposed in paper [11]. The clustering algorithms that use edge zeroing are presented in the paper [6]. The use of the algorithm Dominant Sequence Clustering (DSC) [5] is proposed in this report due to its further consideration on the graph structure (better decisions on fork and join structures). Moreover, Kim and Browne are proposed linear clustering algorithm with the complexity of $v(v+e)$ [14]. The clustering algorithm that is proposed by Wang Gajski [10] has the complexity of $v^2 \log v$ and the Sarker’s algorithm [6] has the complexity of $e(v+e)$. The complexity of DSC algorithm, which is $(v + e)\log(v)$, is the best suited one for the large applications.

Based on research hypothesis and the literature study, the following solution directions (SD) are proposed:

**SD1-Using the existing structure of the application, restrict the deployment freedom of the scheduler by giving fine grained deployment hint:**
Note that using the Y-chart modeling gives the opportunity of separation of concerns for each level. That means changing the deployment part does not affect any other design levels. Each task has a specified list of resources in which the task can be deployed to. This list can be modified. For instance if the list of resources of a task is \{pu1, pu2, pu3\}, the list can be modified to \{pu1, pu2\}. In manual and automatic deployment techniques presented below, the deployment is done by modifying the list of resources.

**SD1.1-Manual Deployment:**
As it is shown in Figure 1.2, an application is exposed to a graph refinement in order to add the timing information of tasks as well as coarse grained deployment information. The coarse grained deployment information consists of a list of possible processing units on which a task is allowed to be deployed. Manually enforcing a deployment choice implies that this
list of possible processing units that a task can be deployed will be reduced.

SD1.2-Automatic Deployment:

In this part, the deployment lists of the tasks explained previously are modified automatically. The first scheduling step is used to determine the deployment of tasks and the second scheduling step is to actually put the tasks into an order so that the timing requirements are satisfied.

SD2-Computing Application Structure:

A new clustering algorithm specific for exploiting the parallelism is applied on the application graphs. The algorithm produces concurrent clusters. This clusters need to be deployed onto limited number of processing units. For the deployment of the clusters, an automatic deployment strategy is applied and compared with the previous automatic deployment strategy that is proposed in the previous solution direction.

SD2.1-Automatic Deployment:

A combined deployment approach is used to deploy the clusters that is obtained from the application structure computation. Bin packing approach [7] is used to ensure the processing units are uniformly distributed. It is combined with the hierarchical cluster grouping approach [13] to deploy the small clusters so that the small clusters are deployed with their most communicating clusters.

The flow of the report is as follows. In Chapter 2, the scheduler and its heuristics are explained. The details about the manual deployment approach SD1.1 are presented in Chapter 3. The automatic deployment approach SD1.2 is introduced in Chapter 4. In Chapter 5, the algorithmic structure detection and automatic deployment approach SD2 is presented. Finally, the conclusion of this thesis project is given in Chapter 6.
Chapter 2

Scheduler

The increase in the number of cores does not increase the frequency evenly. Therefore, in order to increase the performance, the parallelism present in the applications needs to be exploited. When the number of tasks increases, the communication overhead may become more significant. Therefore the deployment decisions made by the scheduler are critical to the performance of the multi-core system [15]. Optimal scheduling of multi-core systems is an NP-hard problem. It can be reduced to a partitioning problem which is NP-complete [9]. Therefore, the scheduler uses heuristics to make its decisions. In this chapter, the heuristics of the scheduler are discussed. The current single-core and multi-core scheduling algorithms used in CARM are presented. As a preliminary, inter and intra worker dependencies are introduced to better understand the scheduling algorithms.

2.1 Preliminaries

In this section the definitions of some terms that will be used later in the report are given to better understand the concept.

**Application:** An application is a Synchronous Homogeneous Data Flow graph in which the nodes of the graph corresponds to the tasks and the edges in the graph corresponds to control and data dependencies between the tasks. The application graph is exposed to some transformations as it is explained in Chapter 1 in Figure 1.2 in order to obtain the schedule.

**Dependency:** A dependency \((a, b)\) denotes that task \(b\) can start its execution only after the completion of task \(a\).

**Dependency Task Graph:** A dependency task graph is a directed graph
representing the tasks and the dependencies of these tasks towards each other. **Execution Time:** The execution time is the time that a task requires to complete its execution on the specific platform that it is mapped to. The execution time is assumed to be fixed in the scope of this work. **Completion Time:** Completion Time is the time at which a task will complete its execution. **Feasible schedule:** A feasible schedule is a schedule that meets its latency requirements. **Makespan:** Makespan is the total length in time of a schedule. **IO delay:** The time takes to complete a single IO operation. **Deadline:** Deadline of a task is the time at which the execution of a task should be completed.

### 2.2 Inter-worker Dependencies

Workers are the entities that perform computations (abstracting from the real hardware). Processing units are the computational units within a worker. A worker can consist of one or more processing units. The dependencies between tasks mapped on different workers are called inter-worker dependencies. After the completion of a task, the communication of sample data and synchronization is required. These dependencies are implemented by W2W (Worker-to-Worker) tasks. Figure 2.1 shows the implementation of these dependencies. After the deployment of SGs onto Workers is known, W2W producer and consumer tasks are added by a graph refinement as shown in Figure 1.2.

![Figure 2.1: Inter-worker dependencies](image-url)
2.3 Intra-worker Dependencies

Intra-worker dependencies are the dependencies between the tasks deployed on different PUs. Within a worker, tasks communicate via shared memory and therefore there is no need for a communication mechanism. However, the data flow semantics of servo applications must be respected. In a single PU worker, task synchronization is enforced by the schedule and the sequencer which is an entity corresponding to each PU that counts the number of task completed. In a multi-PU worker, an additional synchronization mechanism is needed for the tasks deployed on different PUs.

The mechanism to implement intra worker-dependencies can be summarized as follows [3]:

Sequencer states: Each PU has a sequencer which has a state that denotes the number of completed tasks. The state is updated after a task has terminated.

 Guards: Each task has a guard which is a collection of state expressions. A state expression is of the form of $S \geq k$ where $k \in \mathbb{N}$. It is used to enforce synchronization. The set of guards of a task together specify the enabling criteria of a task based on the incoming dependencies of a block. A shared variable is set by an update and it is polled by a corresponding guard.

![Figure 2.2: Optimized guard expressions](image)

After having implemented intra-worker dependencies, it was observed that there were more than necessary guards and updates added in the schedule. Therefore an optimization was required. This optimization minimizes the number of required guards and updates through a transitive reduction algorithm.
Intra-worker Dependencies

The transitive reduction of a directed graph G is the directed graph with the smallest number of edges where the reachability relation of the original graph is preserved [4].

The dependencies in the applications after the graph refinement are classified as non-redundant dependencies and the redundant dependencies. Non-redundant dependencies are the dependencies that are necessary to ensure data flow semantics and are not implied by any other dependencies. In Figure 2.2, non-redundant dependencies are represented by dashed black arrows. Redundant dependencies are the dependencies that are implied by non-redundant dependencies so that they can be removed. In Figure 2.2, redundant dependencies are represented by faint gray arrows. By transitive reduction, the redundant dependencies are removed since they are already implied by the non-redundant dependencies.

A guard only needs to be included in the guard set of a task if it corresponds to a non-redundant dependency. An update is only required if a corresponding guards is used. By computing the strongest condition for a task after it has completed, the redundant dependencies are eliminated. The complexity of the algorithm is $N+M*R$, where
-N denotes the number of blocks;
-M denotes the number of dependencies;
-R denotes the maximum number of sequences (PUs) per worker.

Figure 2.2 shows the optimized guard and update expressions through the transitive reduction algorithm. The gray dashed lines represent the redundant dependencies and the black dashed lines shows the non-redundant dependencies. Red state expressions denote the non-redundant states whereas the gray ones denote the redundant state expressions. For instance, the dependency between task A and task G is a redundant dependency since the corresponding guard $s2 \geq 1$ is implied by the dependency between task F and task G.
2.4 Scheduling

The scheduling is performed on a per task basis using list scheduling. 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 earliest-due-date-first heuristic and it is scheduled one of its resources. The start time of the task 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. The gaps may occur when a task cannot be scheduled immediately after the last scheduled task due to task dependencies. The scheduler tries to fill the existing gaps by the partial schedule [1].

2.4.1 Single-PU Scheduling

In single-PU scheduling, the deployment of all tasks is fixed. The due date calculation is required before the scheduling of the tasks. If a task finishes later than its due date it will result in a latency violation in the follow-up tasks. In order to prune the search space resulting in infeasible schedules as soon as possible, tight due dates are required[3].

2.4.2 Multi-PU Scheduling

Unlike single-PU scheduling, in multi-PU scheduling also the deployment of the tasks onto the processing units should be considered, where the deployment of the tasks on workers are still fixed. The required scheduling steps are similar as in the case of single-PU scheduling. The difference is that when an enabled task is selected, the PU within the worker should also be chosen by the scheduler. The rules to choose the most preferable PU are presented in Figure 2.4. The idea behind the preference rules is to deploy the tasks that imply making a trade-off between meeting the task deadlines and adding intra-worker dependencies. The first step is to check the PUs for the earliest possible start time of the task. If there is more than one deployment option, a PU is chosen based on the deployment of the tasks predecessor and the last completing predecessor. If there is no predecessor or if there is still more than one deployment option, then the PU with the lexicographically earliest name is chosen.
The deployment algorithm takes the penalties into account in the start times of the tasks on PUs different from its predecessors. The transitive reduction is performed after scheduling. Hence during scheduling it is not known which are the redundant guards and updates. Due to this limited information, an assumption is made such that each time a task is scheduled on a different resource than its predecessors, a guard-update penalty is present. This might result in taking penalties into account when they are actually not present.

Figure 2.3: Improved algorithm with guard and updates

Figure 2.4: Preference rules for large tasks
The deployment algorithm applies different rules for small and large tasks as shown in Figure 2.3. The aforementioned preference algorithm is still valid for large tasks. However, deploying a small task into a different PU than its predecessor may bring a cost that exceeds its execution time. Therefore, small tasks are deployed on the same PU with their predecessors until the start time of an alternative-PU is less than the resource fill threshold. Figure 2.5 shows the preference rule for the small tasks. The resource fill threshold ensures that the resources are getting filled uniformly. In case the start time of an alternative-PU is larger than the resource fill threshold, the PU with a predecessor is chosen. If there is more than one deployment option then the PU with the last completing predecessor is chosen. If there is no predecessor, the PU with earlier start time is chosen. In case of two PU options, the PU with lexicographically earliest name is picked.
Chapter 3

Manual Deployment

In this chapter, the deployment strategy which exploits the existing structure of the application is presented. By using the existing structure of the applications, which is defined by mechatronic engineers for reasons of understandability of the system in their terms, some block groups are manually deployed into some specific processing units. The algorithm regarding the choice of the block groups is presented in the next section.

3.1 Manual Deployment Strategy

It is mentioned that litho-scanners used in ASML contain many subsystems to be controlled. One example is a wafer positioning module. This module can move in six degree of freedom (6DOF) where 6 DOFs refer to the freedom of movement of a rigid body where X, Y, Z are the translation movements and Rx, Ry, Rz are the rotational movements [8] as it can be seen in Figure 3.1. These DOFs work independently from each other. Therefore, the inherent parallelism can be exploited in the applications.
The textual structure (in terms of block groups) of part of the application is shown in Figure 3.2.a together with the graphical representation in Figure 3.2.b. The X, Y, Z, Rx, Ry and Rz block groups are defined at a certain hierarchical level of the application.

By exploiting the structure of the control block groups as explained above, the tasks in each 6-DOFs block groups can be deployed into different processing units assuming that they do not have communication dependencies.

The diagram presented in Figure 3.3 represents the flow of the implementation of this method. The 6-DOFs block groups in the input DAG are
manually deployed to specific PUs. The fine grained deployment can be seen in Figure 3.4. The DAG with the restricted deployment options is given to the scheduler and the scheduler creates the schedule.

```
worker: node1

blockMapping {
    blocks:[XPM1, Ctrl X]
    processingUnits: x1
}

blockMapping {
    blocks:[XPM1, Ctrl Y]
    processingUnits: x2
}

blockMapping {
    blocks:[XPM1, Ctrl Z]
    processingUnits: x3
}

blockMapping {
    blocks:[XPM1, Ctrl Rx]
    processingUnits: x4
}

blockMapping {
    blocks:[XPM1, Ctrl Ry]
    processingUnits: x5
}

blockMapping {
    blocks:[XPM1, Ctrl Rz]
    processingUnits: x6
}
```

Figure 3.4: Fine-Grained Deployment of the 6DOF Block Groups
3.2 Results

In this part, the results obtained from the fine grained deployment are presented. The approach explained in the previous section is applied to the stages stack as well as POB.

3.2.1 Case 1: POB mirror network

The latest wafer scanner systems (EUV/NXE) make use of UV light. The Projection Optics Box (POB) is the enclosure which contains the optical components between the reticle and the wafer stage that holds the wafers to be exposed. The POB models contain 1878 tasks and 1540 dependencies [1]. The fine grained deployment is applied to POB (Projection Optics Box) stacks. Figure 3.5 and Figure 3.6 show the original make-span and the improved make-span by fine grained deployment approach introduced in this chapter respectively. While the original make-span in Figure 3.5 is 30.7 units of time, the make-span obtained with the fine grained deployment approach is 26.7 units of time. The make-span obtained from the POB stacks justifies our research hypothesis regarding the improvement of the IO delay with the fine grained deployment. Note that the results of the POB stack are obtained by the domain experts with additional modifications under their additional knowledge of the application. The architecture of the application is modified. Moreover, some tasks are duplicated to avoid the communication overhead between processing units. [2].
Results

Figure 3.5: The original schedule of the POB stack

Figure 3.6: The schedule of the POB stack obtained from the proposed approach
3.2.2 Case 2: Stages Stack

The wafer stage is the responsible part for movement of the wafers. Two wafers can be processed at a time. When the accurate positioning and the orientation of the one wafer is being measured, an exposure can be applied to another wafer. This requires the synchronous movements to transfer the wafer within the system. Therefore, a large number of sensors and actuator are used. The complete wafer stage application contains 4301 tasks and 4095 dependencies [1].

The fine-grained deployment approach is also applied to the stages stacks. Figure 3.7 shows the original schedule of the application. The make-span of this schedule takes 38.692 units of time. Figure 3.8 shows the proposed manual deployment results (38.819 units) and Figure 3.9 shows the manual deployment of the 6DOF blocks onto 3 cores instead of 6 which is suggested by the domain experts (resulting in 38.557 units of time). The improvement obtained from this approach is quite small and it can hardly be observed from the figures.

![Figure 3.7: The original schedule of the stages stack](image-url)
Figure 3.8: The schedule of the stages stack obtained from the proposed approach

Figure 3.9: The schedule of the stages stack with the additional application knowledge

3.3 Conclusion

It is observed that with the additional knowledge of domain experts in the POB application, it is possible to have improvements in the make-span with
the manual fine grained deployment approach [2]. This shows that the man-
ual fine grained deployment confirms the hypothesis that exploiting the struc-
ture of the applications can indeed improve timing performance. However,
without sufficient domain knowledge it appeared to be hard to achieve a suf-
ficient performance improvements. Employing manual deployment strategy
for very large applications is concluded as:

- time consuming
- error-prone
- application specific
- needing complete knowledge of each application

Due to the above drawbacks, the manual fine grained deployment approach
is concluded to be inadequate. Therefore, it is proposed that we utilize
the existing structure in the applications as designed by domain experts in
combination with an automatic deployment strategy.
Chapter 4

Automatic Deployment

In this chapter, an alternative deployment strategy which exploits the existing structure of the application is presented. By using the existing structure of the applications, the hierarchical inner most block groups are automatically enforced to be deployed into some specific processing units. In the next section, the algorithm followed by the automatic deployment strategy is presented.

4.1 Automatic Deployment Strategy

The scheduler takes decisions based on local application knowledge during deployment. From the example presented in the previous chapter, it can be seen that by exploiting more structural information of the application, a better schedule can be obtained.

The applications consist of hierarchical block groups. The tasks and related dependencies are located within the block groups. The hierarchical structure implies that the tasks within the inner most block groups have the most common relations in terms of control engineering aspect. Moreover, the dependencies defined in each level of the hierarchy are related dependencies of the current level or the inner levels in the hierarchy. These hierarchical structure may contain the most dependent tasks through the inner levels in the hierarchy.

Due to hierarchical structure of the applications, it is proposed that deploying the tasks in the inner most block groups onto same processing units can reduce the intra-worker dependencies and consequently can improve the make-span and the IO delay. The assumption under the automatic deployment approach is that if the structures of the block groups in the inner most

...
Automatic Deployment Strategy

hierarchy are as shown in Figure 4.1, then deploying the tasks inside these block groups together onto same processing units improves the makespan.

Let’s consider the same example graph as presented in Figure 1.2 under the aforementioned assumption which is shown in Figure 4.1. By enforcing the scheduler to deploy the tasks within these inner most block groups onto same processing units, the deployment can be obtained automatically.

![Figure 4.1: The possible structure of the block groups](image)

The diagram presented in Figure 4.2 represents the flow of the implementation of the automated deployment. The first step in the implementation flow represents the task aggregation part which has the input of DAG and applies transformation on this DAG. The inner most block groups are aggregated to tasks with this transformation. Figure 4.3 shows the task aggregation on a dependency graph. The upper left block shows the hierarchical blocks with graph representation on the upper right part. The inner most hierarchical blocks are transformed to super tasks as it is shown on the lower left block together with the graph representation on the lower right part in Figure 4.3. The output of this transformation is scheduled which is represented in the second step in the implementation diagram. The resultant schedule has the deployment information of the aggregated tasks. This deployment information regarding the inner most block groups tasks is written back to the original DAG (the third step) and the DAG with the restricted deployment information is scheduled (the fourth step). In other words the scheduler is fed with the restricted deployment information in a similar way to the approach in Chapter 3 so that global structure of the application is considered.
Considering the application structure that is given in Figure 4.1, the automatic deployment approach is verified by testing it in the scheduler considering that each task has a execution time of $3 \times 10^{-7}$ units of time and guard and update has the penalty of $3 \times 10^{-8}$ and $1.65 \times 10^{-7}$ units of time respectively. Figure 4.4 shows the make-span of the scheduler which has $2.58 \times 10^{-6}$ units of time. Figure 4.5 is the resultant make-span after applying automatic deployment algorithm. The resultant make-span takes $2.19 \times 10^{-6}$ units of time. The verification shows that if similar structures as presented in Figure 4.1 exist in the input applications used in ASML, the automated deployment algorithm can improve the make-span.
4.2 Results

In order to test the automatic deployment approach, it is applied to the short stroke servo groups in the stages stack, which is introduced in Chapter 3. The make-span of short stroke servo groups obtained from the current scheduler is presented in Figure 4.6. The make-span takes 39.144 units of time. In Figure 4.7, the make-span of the automatic deployment approach is shown for the same servo groups and it has the resultant make-span of 40.272 units of time. It is observed that with the automatic deployment strategy the make-span is even increased. Since the method uses the idea of restricting the scheduler’s
deployment decision, increased make-span can be concluded as the restriction of the scheduler should be relaxed. For this purpose, some of the inner most block groups are skipped during the automatic deployment. The inner most block groups that are in the first hierarchy are decided not to be in use of restricting the scheduler’s deployment decisions. This relaxed automatic deployment approach is also applied to the same input stacks and indeed there observed some improvements in the make-span. Figure 4.8 shows the resultant make-span after applying the relaxed automated deployment which has 38.9 units of time.

Figure 4.6: The original schedule

Figure 4.7: The schedule obtained from the automatic deployment approach
It can be seen that the improvement obtained from automated deployment approach is not significant. The possible reason is that the existing structure of the block groups is not best suited for the proposed approach which aims to exploit the absence of communication between specific parts of the application. There are also some issues related to proposed automatic deployment approach which is presented in the next section.

4.3 Drawbacks

The automatic deployment algorithm is proposed under the assumption of that the inner most hierarchical block groups contains the most dependent tasks as it is presented in Figure 4.1. If the other structures are considered such as in Figure 4.9, the algorithm may not give an improved make-span but even can increase it. For this purpose, the proposed structure in Figure 4.9 is also tested in the scheduler by applying the automatic deployment strategy. The resultant make-span is presented in Figure 4.10. As it can be seen from the resultant schedule, the make-span has $3.375 \times 10^{-6}$ units of time which is even longer than the original scheduler’s make-span ($2.58 \times 10^{-6}$ units of time) presented in Figure 4.4. This example structure concludes that the automatic deployment approach is strongly application (structure) specific.
Another issue that may rise up with the proposed automatic deployment is that the aggregation of the tasks in the first step of the approach may bring cycles to the graph. Let's consider the example structure that is shown in Figure 4.11. Employing the automatic deployment strategy transforms the structure to that is shown in Figure 4.12 where the block groups 1 and 2 are aggregated to tasks 1 and 2. As it can be seen, this structure is cyclic. However, it is stated before that the input of the scheduler has to be a directed "acyclic" graph. Therefore, the proposed automatic deployment strategy may even create such situations that prevents making use of the scheduler.
4.4 Conclusion

In this chapter of the report, the automated deployment strategy is presented. It is verified that the automated fine grained deployment algorithm confirms the hypothesis that exploiting the structure of the applications can give improved schedule. However, it has some drawbacks which makes the approach insufficient. First of all, it is highly dependent on the structure of the applications. Moreover, it may even be obstacle to the use of scheduler in case of the cycle creation. Therefore, the automatic deployment strategy is concluded as:

- application specific
- insufficient to make full use of the approach

Due to these drawbacks, it is proposed to refine the structure of the applications algorithmically rather than exploiting the existing structure. In the next chapter, the approach of detecting the structure of the applications algorithmically is presented.
Chapter 5

Algorithmic Detection of the Application Structure and Automatic Deployment

The approaches explained in the previous chapters that exploit the existing structure of the control blocks show that the existing structure of the applications is not sufficiently prone to exploit parallelism. The structure of the applications is built by mechatronics engineers for reasons of control engineering. Therefore, a better approach seems to detect the structure of the applications algorithmically to exploit the parallelism and take advantage of it to reduce the intra-worker dependencies.

In this chapter, the application structure detection algorithm is presented namely Dominant Sequence Clustering Algorithm (DSC) [6] [5]. Moreover, automatic deployment alorithm is proposed for the deployment of the clustered tasks.

5.1 Implementation

In this section the implementation details of the structure computation and automatic deployment is presented. The diagram presented in Figure 5.1 represents the flow of the implementation of algorithmic structure detection approach. The tasks in the input DAG are clustered by the Get Clusters block. Get Cluster block performs the DSC algorithm, which is presented in the next chapter. The output of the Get Clusters step is a DAG with possible 1-to-many Cluster-to-worker deployment. Since the coarse grained deployment of the block groups is fixed, the clusters which contains the tasks
from different workers need to be split which is shown as the Split Clusters step. After Split Clusters step, each cluster contains the tasks which have the same coarse grained deployment. The number of clusters obtained in the clustering step may be higher than the number of processing units of the worker. Therefore, a deployment strategy for the clusters is necessary. This is presented in the Define Cluster deployment step. After having the deployment information for the clusters, the DAG is scheduled. Note that there are other inputs than the DAG in every stage in Figure 5.1. These inputs are necessary for the algorithms in the related steps since DAG does not contain these informations. The clustering algorithm needs to know the communication overhead when the dependent tasks are deployed onto different processing units. The Split Cluster algorithm uses the coarse grained deployment information. The cluster deployment algorithm needs the number of processing units in the workers.

The clustering and the cluster deployment algorithms are presented in the next two sections.

![Figure 5.1: The implementation flow of the application structure detection algorithm](image)

### 5.1.1 Dominant Sequence Clustering Algorithm

The Dominant Sequence Clustering (DSC) algorithm is based on a DAG representation of the applications where each node represents a task and each weighted edge represents the dependency between two nodes. The weights on the edges are the synchronization times due to dependent tasks deployed on different processing units.

Initially, each task is assumed to be in a different cluster and each cluster is assumed to be deployed onto a different resource. A concept of parallel time (PT) is defined as the make-span of the application. The designated goal is to cluster the tasks which should be deployed onto same processing
Implementation

units. This is achieved by refining the previous clustering in each step of the clustering algorithm. An edge zeroing algorithm [11] is used for this purpose. It merges two clusters and zeros the communication overhead in between. The decision to zero an edge and to cluster the corresponding tasks is given by calculating the PT at each step. At each step one edge should be examined. Initially all tasks are defined as unscheduled. A task is scheduled if it is a head node (no incoming dependencies) or if the task is examined. A task is free if all its predecessors are scheduled. A task can only be chosen to examine if it is free. After choosing the task to examine, the algorithm checks if the edge can be zeroed. It is accepted if the zeroing does not increase the PT. Another additional condition to zero the edge is if the task to be zeroed can start earlier with this zeroing. If both conditions hold then the zeroing is accepted. The decision of which edge to analyze is decided by the dominant sequence. A dominant sequence (DS) is defined as the critical path which is the longest path in the graph. Note that after each iteration, the DS changes due to zeroing an edge. The first unexamined edge in the current DS is chosen as the edge to examine at each iteration.

The algorithm stops after analyzing all the edges in the graph. The number of clusters obtained from the DSC algorithm gives the degree of concurrency in the application. However, this algorithm assumes unbounded number of resources. Therefore, additional implementation is needed to reduce the number of clusters since the number of resources is limited.

If we consider the same example in Figure 1.4 where each task has execution time of 1 units of time and the each edge has the weight of 0.3 units of time, the DSC algorithm starts from the head node 0 and finds 4 critical paths with 6.2 units of time. The node list of the critical paths are \{0,1,2,4,9\}, \{0,1,3,4,9\}, \{0,5,6,8,9\} and \{0,5,7,8,9\}. Starting from the first critical path \{0, 1, 2, 4, 9\}, it checks if node 1 is free. Since it is free, it checks if the current critical path reduces with the clustering node 0 and node 1 and if the start time of node 1 reduces. Clustering node 0 and 1 gives 5.9 units of time for the current critical path and node 1 can start at time 1 instead of 1.3. Therefore zeroing is accepted. The algorithm traverse on each critical path until there is no node to analyze and the resultant clusters are as follows \{0,1,2\}, \{3,4\}, \{5,6\} and \{7,8,9\} as shown in Figure 5.2.
Assuming that the number of processing units are unbounded, the algorithm suggests to deploy each cluster onto different processing unit where the number of clusters is 4. The deployment using 4 processing units is shown in Figure 5.3.
5.1.2 Combined Approach for Deployment

In order to deploy the clusters to a limited number of processing units, the combined deployment strategy, originating from bin-packing [7] and Hierarchical Cluster Grouping [13] algorithms, is applied. The bin-packing is used to have a balanced load in the processing units. The size of a cluster is the total execution time of the tasks in that cluster. Starting from the large clusters, the deployment is applied according to bin-packing algorithm until the number of residual clusters is less than or equal to the number of the processing units. When the number of residual clusters is less than or equal to the number of processing units, the hierarchical cluster grouping algorithm is applied. Hierarchical cluster grouping deploys the most communicating clusters together. The pseudo algorithm for combined approach for deployment is shown in Figure 5.4.

Figure 5.4: Combined Approach for Deployment

For the same clusters that are obtained from DSC algorithm, let's consider that the number of processing units is 2. Employing the combined deployment approach deploys the large clusters \{0,1,2\} and \{7,8,9\} onto processing units pu1 and pu2 respectively. Then considering the hierarchical clustering approach, it deploys the cluster \{3,4\} onto pu1 and cluster \{5,6\} onto pu2.

Figure 5.5 shows the schedule obtained from the proposed combined deployment algorithm. The make-span of the schedule is $2.16 \times 10^{-6}$ units of time. This approach is compared with the suggested deployment in automatic deployment algorithm using existing application structure in Chapter 4. Figure 4.5 shows the schedule from this suggested deployment in Chapter 4 and the make-span of this schedule is $2.19 \times 10^{-6}$ units of time. The comparison of the two automatic deployment algorithms, one uses existing application structure and the other computes the application structure, shows that detecting structure of the applications can even reduce the make-span further.
Figure 5.5: The schedule after DSC algorithm with restricted number of PUs

5.2 Conclusion

The evaluations show that the algorithm for detecting the structure of the applications in order to exploit the parallelism improves the make-span as well as the IO delay and the sampling frequency. Applying DSC algorithm helps not only to improve the make-span but also to reveal the degree of concurrency in the applications. Moreover, unlike the approaches presented in the previous chapters, detecting the structure of the application is not application specific and does not depend on the domain knowledge. Therefore, employing structure detection would benefit IO delay and sampling frequency of the servo control applications.
Chapter 6

Conclusion

The wafer scanners developed at ASML contain many servo control systems with tight hard real time requirements. Applications need to run at high rates and satisfy hard latency requirements between sensing and actuation in order to meet the required machine performance.

The end goal of this project is to reduce IO delay and end-to-end latency of the control applications on multi-core platforms. This in turn results in increase in the control frequency. Current scheduler is designed lacking the global application knowledge. To overcome this issue, deployment strategies, that exploit global application knowledge, are investigated to reduce pu2pu synchronization overhead.

A manual deployment strategy which exploits the inherent parallelism of 6 DOF block groups is investigated in Chapter 3. The evaluations showed that manual deployment is application specific, error prone and time consuming and depends on the domain knowledge.

Automatic deployment strategy which exploits the structure of the applications is examined in Chapter 4. The results showed that this approach is also application specific and insufficient to make full use of the approach. Moreover, the structure of the applications which are designed by the mechatronic engineers for the understandability of the system in their terms is concluded not for exploiting the parallelism for multi-core systems. Therefore the structure of the applications needs to be detected algorithmically.

Algorithmic structure detection strategy is proposed in Chapter 5. The evaluations concluded that this strategy is not application specific and does not require domain knowledge. Therefore, employing structure detection is promising to minimize the IO delay of the servo control applications.

For the future work of this project, the results of the automatic structure detection and deployment strategy for the ASML systems should be
analyzed. After employing the DSC algorithm, in case there is a need for separation of the clusters if they contain tasks with different coarse grained deployment, further discussion will arise regarding the coarse grained deployment. Changing the coarse grained deployment, which is fix, may need to be considered accordingly.
Bibliography


