Optimizing the Placement of Applications in Autonomous Vehicles

Tobias Kain

Automated Driving, Volkswagen AG, Germany. E-mail: tobias.kain@volkswagen.de

Hans Tompits

Institute of Logic and Computation, Technische Universität Wien, Austria. E-mail: tompits@kr.tuwien.ac.at

Julian-Steffen Müller

Automated Driving, Volkswagen AG, Germany. E-mail: julian-steffen.mueller@volkswagen.de

Maximilian Wesche

Automated Driving, Volkswagen AG, Germany. E-mail: maximilian.wesche@volkswagen.de

Yael Abelardo Martinez Flores

Automated Driving, Volkswagen AG, E-mail: yael.abelardo.martinez.flores@volkswagen.de

Hendrik Decke

Automated Driving, Volkswagen AG, Germany. E-mail: hendrik.decke@volkswagen.de

Full vehicle autonomy requires the interplay of numerous software applications. These applications are distributed amongst the computing nodes installed in a vehicle. Due to various requirements and constraints concerning, e.g., resources and safety, determining the assignment between applications and computing nodes is a complex task. This problem is an instance of the so-called *application-placement problem*, which is a well-known challenge in other areas like cloud computing. However, for handling the application-placement problem in the context of autonomous vehicles, only few approaches exist so far. In this paper, we introduce a framework for addressing the application-placement problem for the case of autonomous driving using a linear optimization approach. In particular, we introduce a software architecture for dealing with the application-placement problem along with an embedded solver, called APD. The latter is realized in terms of OR-Tools, an optimization framework developed by Google, to define a set of linear constraints and various linear optimization goals. Furthermore, our architecture also includes a component for deploying the resulting placement plans as well as a parallel-threads heuristic to improve computation time.

Keywords: autonomous vehicles, system reconfiguration, application placement, optimization.

1. Introduction

Since the introduction of motorized vehicles in the early 20th century, the task of operating these machines is the responsibility of the driver. In recent years, however, vehicles have been gradually equipped with an increasing number of driver-assistance systems. Modern vehicles are, among other tasks, capable of keeping the distance to a preceding vehicle, switching lanes on highways, and reversing into a parking space autonomously.

Due to the rapid technological development, it is feasible to automate the task of driving in the near future entirely. Such autonomous vehicles, which are also referred to as *SAE Level 5 vehicles* (SAE International (2018)), can enable many positive effects. For instance, they can induce an improved traffic flow (Fernandes and Nunes (2012)), result in a reduced amount of emissions (Morrow et al. (2014)), or provide mobility for elderly or disabled individuals (Fagnant and Kockelman (2015)). Furthermore, it is assumed that autonomous vehicles lead to safer roads (Maurer et al. (2016)).

To operate such autonomous vehicles, numerous software applications (Jo et al. (2014)), including, for instance, perception, planning, and vehicle control services, are necessary. These applications are distributed among the computing nodes that are installed in a vehicle. All these applications have different requirements concerning, e.g., the needed computing resources, libraries, and safety levels. However, the computing nodes offer only a limited set of resources. Therefore, it

Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference
Edited by Piero Baraldi, Francesco Di Maio and Enrico Zio
Copyright © ESREL2020-PSAM15 Organizers.*Published by* Research Publishing, Singapore.
ISBN: 978-981-14-8593-0; doi:10.3850/978-981-14-8593-0

is not trivial to determine an assignment between applications and computing nodes. This problem is referred to as the *application-placement problem*.

The relevance of the application-placement problem is not limited to the area of autonomous vehicles. Indeed, the placement of applications to computing nodes is a well-studied topic in other areas. In particular, research in cloud and edge computing has addressed this problem in various publications, where the focus of these works is on optimizing different properties including, e.g., energy consumption (Li et al. (2009)), network traffic load (Alicherry and Lakshman (2012)), or resource utilization (Ben Jemaa et al. (2016)).

In this paper, we introduce a software architecture for handling the application-placement problem in the domain of autonomous driving. Our approach allows not only to solve the application-placement problem but also facilitates the deployment of the resulting placement. Initiating the computation of a new application placement can have different causes: for instance, the occurrence of a failure or a change in the driving situation can trigger the computation of a new application placement.

For computing solutions of the application-placement problem, which is a complex task as determining whether a solution exists is NP-hard, we implemented a solver, `APD`, based on formulating the problem in terms of a set of linear constraints, together with a linear optimization goal, and using the constraint-optimization framework developed by Google. Using a particular test setup, we show that the solving time of `APD` linearly correlates to the problem size in case a solution for the defined problem exists. However, in case no solution exists, the correlation between the solving time and the problem size is exponential, reflecting the inherent NP-hardness of the general application-placement problem. To address this, our framework incorporates a method that allows to define multiple variations of the initial application-placement problem, reflecting different priority levels, which are computed in parallel, whereby the solution space of the varied problems is larger than the solution space of the initial problem, giving rise to an increased possibility of finding valid solutions.

Previous approaches for handling the application-placement problem in the automotive domain include, for instance, the work by Pourmohseni et al. (2017), dealing with a dynamic task migration in case of changing resource constraints, the method of Becker and Voss (2015), who apply a graceful degradation to find a valid mapping after the occurrence of a failure, and the approach of Weiss et al. (2020) in which a graceful degradation and dynamic mapping is combined.

Our approach for handling the application-placement problem differs from these works as it allows the definition of multiple optimization goals, which makes it possible to employ our method in various scenarios, in particular for reconfiguring the system after a failure or after a change in the driving context.

The paper is organized as follows: In Section 2, we introduce the application-placement problem for our setting of autonomous driving. In Section 3, we present a software architecture to handle the application-placement problem and, in Section 4, we discuss our solver `APD` for the application-placement problem. Finally, in Section 5, we conclude the paper and discuss future developments.

2. The Application-Placement Problem

The task of determining the assignment between applications and computing nodes is referred to as the *application-placement problem*. In our setting, the input of this problem is a set A of applications and a set N of computing nodes, and the task is to find a function C, called *configuration*, that maps each application $a \in A$ to exactly one node $n \in N$ such that certain constraints, defined in terms of a set of parameters, are satisfied. Moreover, in order to discriminate among a potentially large number of solutions, we utilize an additional optimization function that specifies which valid node assignment is desired most. Conceivable optimization goals are, e.g., maximizing the number of computing nodes that execute no applications, maximizing the redundancy of a specific class of applications, or minimizing the number of displacements after the occurrence of a failure.

For the application-placement problem that we consider in this paper, we define for each application the following parameters: (i) the memory demand, (ii) the CPU demand, (iii) the software requirements, and (iv) the function that the application implements. We assume that one application implements only one function (e.g., pedestrian recognition, map service, localization service, etc.) and multiple applications can implement the same function. For each function, we define the following parameters: (i) the priority, (ii) the level of redundancy, i.e., the minimum number of applications implementing the same functions that have to be executed, and (iii) the level of hardware segregation. Note that the system architect has to define the values for those parameters at design time. Furthermore, for each computing node, we specify the memory capacity, the CPU capacity, and the installed software.

The constraints a valid solution of our application-placement problem needs to satisfy are the following:

- (C_1) An application has to be executed by exactly one computing node.
- (C_2) The sum of the memory demands of all applications running on a computing

node cannot exceed the memory capacity of that node.

(C_3) The sum of the CPU demands of all applications running on a computing node cannot exceed the CPU capacity of that node.

(C_4) An application runs only on such a computing node which offers the software required by that application.

(C_5) The applications that implement the same function have to run on at least a certain number of distinct computing nodes, i.e., the level of hardware segregation has to be satisfied for each function.

Note that, for safety-critical functions, hardware segregation is essential since it avoids that multiple applications of the same function are affected by the failure of a computing node.

From a computational point of view, solving the application-placement problem is a complex task. Indeed, it is easily seen that the version of the application-placement problem studied by Tang et al. (2007) is a special case of our variant and as the former one is NP-hard, it follows that our formulation of the problem is NP-hard too. In fact, NP-hardness follows from the fact that the *class constrained multiple-knapsack problem* (Shachnai and Tamir (2001)), which is NP-hard, can be encoded by a polynomial-time reduction to the application-placement problem.

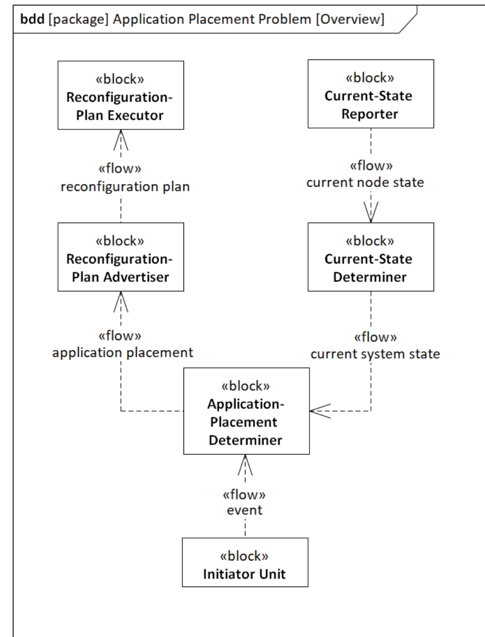

Fig. 1.: A SysML block definition diagram of the system responsible for handling the application-placement problem.

3. A Software Architecture for Handling the Application-Placement Problem

For solving the application-placement problem, we designed a software architecture consisting of several individual components. Our architecture allows for dividing the responsibilities of the different components and thus keeps their complexity reasonable. Furthermore, adaptations of the individual components are in that way easier to maintain.

Overall, we define five different components: (i) a *current-state reporter*, (ii) a *current-state determiner*, (iii) an *application-placement determiner*, (iv) a *reconfiguration-plan advertiser*, and (v) a *reconfiguration-plan executor*.

The current-state reporter and the reconfiguration-plan executor are executed by each computing node installed in a vehicle, whereby the remaining components, on the other hand, are distributed amongst the installed computing nodes. Figure 1 illustrates the components as well as the information flow. Note that in order to increase the reliability, a redundancy and segregation concept for the components responsible for handling the application-placement problem is required. The idea is that the components illustrated in Figure 1 are executed redundantly, whereby the redundant instances are heterogeneous. Furthermore, the redundant instances are distributed amongst the computing nodes installed in the vehicles.

As mentioned before, the current-state reporter is executed by each computing node. The task of this component is to report the current state of the node to the reconfiguration unit. This includes information about the available resources and information about the applications that are currently executed by this computing node.

The information provided by the current-state reporter is used by the current-state determiner to obtain the current state of the entire system, containing knowledge about all available computing nodes, the placement of the applications, as well as the requirements of the applications.

The current system state is used as input for the application-placement determiner, which is the main component and the most complex one involved in the reconfiguration procedure. This component is also the point of entry for the reconfiguration procedure, for which the *initiator unit* notifies the application-placement determiner about the occurrence of an event that requires the computation of a new application placement. Such an initiation for computing a new application placement can have different causes, like, e.g., the occurrence of a failure, the request to start a new application, or the optimization of the system safety.

Once the application-placement determiner computed a new placement, it forwards this plan, as well as the system state that was the basis for the determination, to the reconfiguration-plan advertiser. This component computes, based on the received inputs, a reconfiguration plan, i.e., a set of commands that transfers the current state to a new system state computed by the application-placement determiner. These commands instruct a computing node to either start or stop an application.

The components receiving those commands are the reconfiguration-plan executors. Those components are responsible for executing the reconfiguration commands computed by the reconfiguration-plan advertiser.

4. Application-Placement Determiner Implementation

As mentioned before, the application-placement determiner, which is part of the reconfiguration unit, is responsible for determining the placement of applications. Therefore, this component determines the properties defined by the functions, the requirements of the application, and the system resources. Based on this information, the application-placement determiner computes a new application placement.

In this section, we describe the system `APD`, which is an implementation of an application-placement determiner. The solving approach implemented by `APD` is *integer linear programming* (Schrijver (1986)), a problem formulation which is restricted to linear constraints and objective functions. The framework used to specify our problem is the constraint-optimization framework OR-Tools, developed by Google and available at

`https://developers.google.com/optimization`.

`APD` is specifically designed to reconfigure the system after the occurrence of a hard- or software failure. In particular, the optimization goal implemented by `APD` aims for new application placements that require a minimal number of application displacements so that the reconfiguration actions can be performed fast.

4.1. *Software Architecture of* `APD`

As illustrated in Figure 2, the structural design of `APD` consists of the following three components: (i) the *input preprocessor*, (ii) the *application-placement solver*, and (iii) the *solution processor*.

The input preprocessor is responsible for transforming the current state information into a valid application-placement problem definition. Based on that, the input preprocessor creates a thread of four versions (including the original one) of the given problem definition, which have different

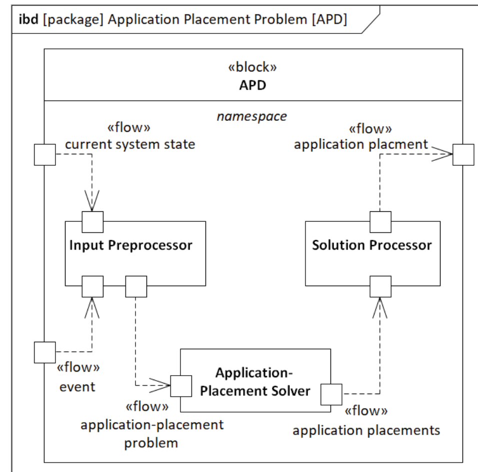

Fig. 2.: A SysML internal block diagram, showing the structural design of `APD`.

solution-space sizes and are processed in parallel. The application-placement solver then attempts to solve these four problems in parallel. Finally, the solution processor selects the most desired solution and forwards it to the reconfiguration-plan advertiser. A more detailed discussion of this parallel processing is given in Subsection 4.4.

4.2. *Formalization of the Application-Placement Problem*

The application-placement solver determines an application placement based on a given system state, S, which is represented by a 9-tuple of the form $(A, F, N, R, \Phi, \Omega, \phi, \omega, \pi)$, whose elements are defined as follows:

- A is the set of applications;
- F is a set of functions, where each application $a \in A$ belongs to exactly one function $f \in F$;
- N is the set of computing nodes;
- R is the *placement restriction function*, which specifies whether a computing node $n \in N$ fulfills all software requirements of $a \in A$, defined by setting $R(n, a) = 1$ if n fulfills the requirements of a and $R(n, a) = 0$ otherwise (we write $R_{a,n}$ for $R(n, a)$ in what follows);
- Φ is the function which assigns each $n \in N$ its memory capacity $\Phi(n) = \Phi_n$ in megabytes;
- Ω is the function which assigns each $n \in N$ its CPU capacity $\Omega(n) = \Omega_n$ in cycles per second;
- ϕ is the function which assigns each $a \in A$ its memory demand $\phi(a) = \phi_a$ in

megabytes;

- ω is the function which assigns each $a \in A$ its CPU demand $\omega(a) = \omega_a$ in cycles per second; and
- π is the function assigning each $f \in F$ the minimum number $\pi(f) = \pi_f$ of computing nodes F has to run on, i.e., the level of hardware segregation of f.

Based on the system state S determined by the input preprocessor, the application-placement solver computes a configuration function C which specifies whether a computing node $n \in N$ executes an application $a \in A$ by setting $C(n, a) = 1$ if n executes a and $C(n, a) = 0$ otherwise. Similar to the representation of the placement restriction function, we will use $C_{a,n}$ to stand for $C(n, a)$.

Since all elements of the configuration function C are binary, at most $2^{|A|\cdot|N|}$ potential solutions exist. Valid solutions, however, must satisfy constraints (C_1)–(C_5) from Section 2, which can be modelled in terms of the following linear constraints:

Constraint (C_1):

$$\forall a \in A : \sum_{n \in N} C_{a,n} = 1. \quad (1)$$

Constraint (C_2):

$$\forall n \in N : \sum_{a \in A} \phi_a \cdot C_{a,n} \leq \Phi_n. \quad (2)$$

Constraint (C_3):

$$\forall n \in N : \sum_{a \in A} \omega_a \cdot C_{a,n} \leq \Omega_n. \quad (3)$$

Constraint (C_4):

$$\forall a \in A, \forall n \in N : \text{if } R_{a,n} = 0, \text{ then } C_{a,n} = 0. \quad (4)$$

Constraint (C_5): To express this constraint linearly, we introduce an ancillary variable $h_{a,n}$ for each application $a \in A$ and computing node $n \in N$, which is defined as follows:

$$\forall f \in F, \forall n \in N : h_{f,n} = \begin{cases} 1, \text{ if } \sum_{a \in f} C_{a,n} \geq 1, \\ 0, \text{ otherwise.} \end{cases}$$

With the use of these variables, we can formalize constraint (C_5) by the following four linear expressions:

$$\forall f \in F, \forall n \in N : h_{f,n} = 0 \text{ or } h_{f,n} = 1, \quad (5)$$

$$\forall f \in F, \forall a \in f, \forall n \in N : C_{a,n} \leq h_{f,n}, \quad (6)$$

Table 1.: The configuration function C.

C	n_1	n_2	n_3	n_4
a_1	0	1	0	0
a_2	0	0	1	0
a_3	0	0	0	1
a_4	1	0	0	0
a_5	1	0	0	0
a_6	1	0	0	0

$$\forall f \in F, \forall n \in N : \sum_{a \in f} C_{a,n} \geq h_{f,n}, \text{ and} \quad (7)$$

$$\forall f \in F : \sum_{n \in N} h_{f,n} \geq \pi_f. \quad (8)$$

Since the modeling of (C_5) is not straightforward, we give an example to clarify its meaning.

Example 1. Consider a system state S which defines two functions, six applications, and four computing nodes, as represented by $F = \{f_1, f_2\}$, $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, and $N = \{n_1, n_2, n_3, n_4\}$. The applications a_1, a_2, and a_3 belong to functions f_1 and the remaining applications belong to f_2, where the hardware segregation level of both functions is 3, i.e., $\pi_{f_1} = \pi_{f_2} = 3$. Furthermore, the system state S defines a configuration function C, as specified in Table 1.

We first consider function f_1: Since

$$\sum_{a \in f_1} C_{a,n_1} = 0,$$

(7) causes that $h_{f_1,n_1} = 0$, and (6) yields that

$$h_{f_1,n_2} = h_{f_1,n_3} = h_{f_1,n_4} = 1$$

since $C_{n_2,a_1} = C_{n_3,a_2} = C_{n_4,a_3} = 1$. As a result, also (8) holds since

$$\sum_{n \in N} h_{f_1,n} = 3 \geq \pi_f = 3.$$

Next, we show that (C_5) does not hold for function f_2: As $C_{a_4,n_1} = 1$ (note that also $C_{a_5,n_1} = C_{a_6,n_1} = 1$) and (6) has to hold, $h_{f_2,n_1} = 1$ holds. Since $C_{a_x,n_y} = 0$ (for $x \in \{4, 5, 6\}$ and $y \in \{2, 3, 4\}$) and (7) has to hold,

$$h_{f_2,n_2} = h_{f_2,n_3} = h_{f_2,n_4} = 0$$

holds. As a result, also (8) does not hold since

$$\sum_{n \in N} h_{f_2,n} = 1 \ngeq \pi_f = 3.$$

□

In view of this example, we can state the following result:

Theorem 1. *Given a system state*

$$S = (A, F, N, R, \Phi, \Omega, \phi, \omega, \pi)$$

satisfying conditions (1)–(8) and a configuration function C, for any $a \in A$ and any $n \in N$, the ancillary variables $h_{a,n}$ satisfy the following two conditions:

$$h_{a,n} = 0 \text{ iff } \sum_{i \in a} C_{i,n} = 0 \quad (9)$$

and

$$h_{a,n} = 1 \text{ iff } \sum_{i \in a} C_{i,n} \geq 1. \quad (10)$$

Proof. We first show condition (9). Assume $h_{f,n} = 0$. Towards a contradiction assume

$$\sum_{a \in f} C_{a,n} > 0 \quad (11)$$

(since, by definition, $\sum_{a \in f} C_{a,n}$ cannot be negative, we do not have to consider the case that $\sum_{a \in f} C_{a,n} < 0$). However, (11) implies that there exists some $a_0 \in F$ and some $n_0 \in N$ such that $C_{a_0,n_0} > 0$, and therefore condition (6) is violated.

Conversely, assume now that

$$\sum_{a \in f} C_{a,n} = 0$$

but $h_{f,n} = 1$ (note that condition (5) restricts that $h_{f,n}$ is either 0 or 1). However, in this case, (7) is violated since $0 = \sum_{a \in f} C_{a,n} < h_{f,n} = 1$.

Now we prove condition (10). Assume that $h_{f,n} = 1$ but $\sum_{a \in f} C_{a,n} < 1$. The latter implies that $\sum_{a \in f} C_{a,n} = 0$ must hold, which contradicts condition (7).

Conversely, assume that $\sum_{a \in f} C_{a,n} \geq 1$ but $h_{f,n} = 0$ (note again that condition (5) restricts that $h_{f,n}$ is either 0 or 1). $\sum_{a \in f} C_{a,n} \geq 1$ implies that there is some $a_0 \in F$ such that $C_{a_0,n} > 0$. Therefore, condition (6) is violated. □

To discriminate among the different solutions, specifying which solutions are desired the most, we instruct the solver to find an application placement which maximizes the following optimization goal, where $\overline{C}$ represents the current node assignment:

$$\sum_{a \in A, n \in N} C_{a,n} \times \overline{C}_{a,n}.$$

Due to this optimization goal, application placements that minimize the number of application displacements are preferred. We aim for a low number of application displacements since displacements are considered to be time-consuming.

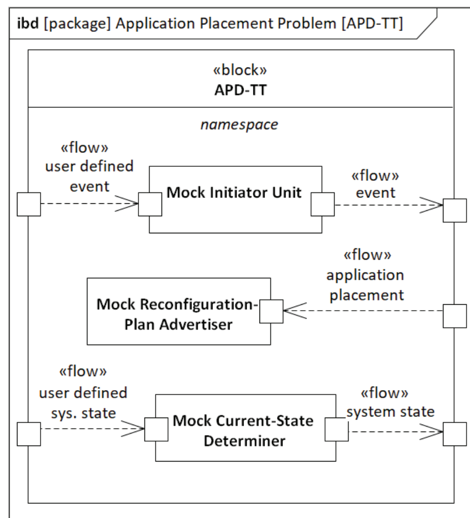

Fig. 3.: A SysML internal block diagram, showing the structural design of APD-TT.

4.3. *Performance Evaluation*

To evaluate the performance of APD, we developed the application-placement determiner testing tool APD-TT, which is a web tool that allows to test the placement behavior of an application-placement determiner in different situations and measure its performance. The idea is that a user defines test cases which specify the current system state in a JSON format as well as the event that triggers the computation of a new application-placement.

As illustrated in Figure 3, APD-TT mocks the behavior of the initiator unit, the current-state determiner, and the reconfiguration-plan advertiser. The actions performed by these components can be controlled and displayed using the AngularJS frontend of APD-TT. The backend of APD-TT is developed in Java and uses the Spring framework.[a] It is responsible for providing the data required by the frontend via REST interfaces. Furthermore, the backend communicates with the application-placement determiner via gRPC.[b]

The computer used to run the performance test was equipped with an Intel Pentium Silver J5005 processor and 8 GB memory. The specifications of this CPU are shown in Table 2.

Since the computational effort of solving the application-placement problem strongly depends on the size of the problem, we defined four problem-size classes for which we fixed the number of computing nodes to four and altered the

[a] https://spring.io/.
[b] https://grpc.io/.

Table 2.: The specification of the processor used for the performance evaluation of APD.

	Intel Pentium Silver J5005
Number of Cores	4
Base Frequency	1.50 GHz
Burst Frequency	2.80 GHz
Cache	4 MB

Table 3.: Average solving time of 1000 test cases that belong to the same problem-size classes. For all test cases a solution exists.

Problem-Size Class	Solving Time
$\lvert N\rvert = 4, \lvert A\rvert = 50$	30 ms
$\lvert N\rvert = 4, \lvert A\rvert = 100$	61 ms
$\lvert N\rvert = 4, \lvert A\rvert = 500$	334 ms
$\lvert N\rvert = 4, \lvert A\rvert = 1000$	823 ms

Table 4.: Average solving time of 1000 test cases that belong to the same problem-size classes. For none of the test cases a solution exists.

Problem-Size Class	Solving Time
$\lvert N\rvert = 4, \lvert A\rvert = 15$	8 ms
$\lvert N\rvert = 4, \lvert A\rvert = 20$	40 ms
$\lvert N\rvert = 4, \lvert A\rvert = 25$	691 ms
$\lvert N\rvert = 4, \lvert A\rvert = 27$	2,457 ms

number of applications. Each problem-size class consists of 1000 randomized test cases.

The average solving time of the different problem-size classes are shown in Table 3. This table shows that the correlation between the number of applications and the average solving time is almost linear. Note that all test cases are designed such that a valid solution exists. However, as not for all application-placement problem instances valid solutions exist, we designed sets of randomized test cases for which this is the case. The results of these test runs are shown in Table 4. These results indicate an exponential correlation between the number of applications and the average solving time in case no solution exists, reflecting the NP-hardness of the general problem. To potentially reduce the computing time, we adopted a parallel-solving heuristic as described next.

4.4. *Parallel-Solving Heuristic*

To increase the chances of obtaining a valid solution, we divide a given set F of functions into several subsets. APD computes these subsets based on priorities in terms of the following four priority classes: $HIGHEST$, $HIGH$, LOW, and $LOWEST$.

Using these priority classes, APD computes the following subsets of applications:

$$F_0 = F = HIGHEST \cup HIGH \cup LOW \cup LOWEST,$$
$$F_1 = HIGHEST \cup HIGH \cup LOW,$$
$$F_2 = HIGHEST \cup HIGH, \text{ and}$$
$$F_3 = HIGHEST.$$

For each of those four subsets, APD starts a thread that computes an application placement that considers all applications that are part of the respective subset. Since $F_3 \subseteq F_2 \subseteq F_1 \subseteq F$ holds, we can assume, as illustrated in Figure 4, that the time required to find a valid application placement is highest for F and lowest for F_3. Furthermore, it holds that in case a valid application placement for subset F_i exists, for $0 \leq i \leq 2$, a solution for subset F_{i+1} exists as well. On the other hand, if for subset F_i no solution exists, we can infer that also for each subset F_{i-j}, for $1 \leq j \leq i$, no solution exists.

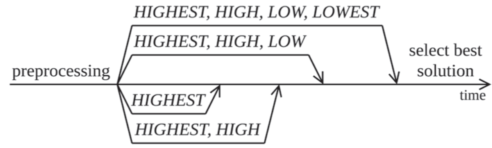

Fig. 4.: Multiple threads for determining the application-placement problem for subsets of applications.

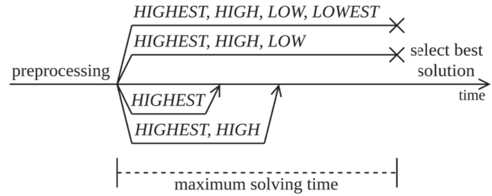

Fig. 5.: Maximum solving time causing two application-placement determiner threads to abort.

After the termination of all threads, APD selects the best available solution, whereby an application placement that considers all applications of subset F_0 is the most desired solution and an application placement that only maps applications of priority class $HIGHEST$ is referred to as the *worst-case solution*.

Besides allowing to find an application placement for the most important applications in case that the system cannot run all applications, this priority-based approach also allows to define an upper bound for the solving time for the application placement. As illustrated in Figure 5, defining a maximum solving time causes that the application determiner threads that cannot find a valid solution within the specified time are aborted.

Defining a maximum solving time is desirable since some safety-critical situations require an application placement within a guarded time rather than a placement that considers all applications.

In case where for none of the four subsets a solution can be found, an emergency system brings the vehicle to a safe stop (Kain et al. (2020)).

5. Conclusion

In this paper, we addressed the application-placement problem for the case of autonomous driving. In particular, we presented a distributed software architecture that allows handling the application-placement problem. Based on this architecture, we implemented APD, a solver for the application-placement problem using linear programming.

For future work, we plan to further improve the performance of the APD solver and integrate it into our approach for a fail-operational system design, called FDIRO (Kain et al. (2020)). This includes also the development of a redundancy concept for the components responsible for handling the application-placement problem to increase the overall reliability. Furthermore, we plan to use APD in our framework for a context-based system architecture for autonomous vehicles (Kain et al. (2020)).

References

Alicherry, M. and T. Lakshman (2012). Network Aware Resource Allocation in Distributed Clouds. In *Proceedings of the 2012 IEEE International Conference on Computer Communications* (*INFOCOM 2020*), pp. 963–971.

Becker, K. and S. Voss (2015). Analyzing Graceful Degradation for Mixed Critical Fault-Tolerant Real-Time Systems. In *Proceedings of the 18th IEEE International Symposium on Real-Time Distributed Computing* (*ISORC 2015*), pp. 110–118.

Ben Jemaa, F., G. Pujolle, and M. Pariente (2016). QoS-Aware VNF Placement Optimization in Edge-Central Carrier Cloud Architecture. In *Proceedings of the 2016 IEEE Global Communications Conference* (*GLOBECOM 2016*), pp. 1–7.

Fagnant, D. J. and K. Kockelman (2015). Preparing a Nation for Autonomous Vehicles: Opportunities, Barriers and Policy Recommendations. *Transportation Research Part A: Policy and Practice 77*, 167 – 181.

Fernandes, P. and U. Nunes (2012). Platooning With IVC-Enabled Autonomous Vehicles: Strategies to Mitigate Communication Delays, Improve Safety and Traffic Flow. *IEEE Transactions on Intelligent Transportation Systems 13*(1), 91–106.

Jo, K., J. Kim, D. Kim, C. Jang, and M. Sunwoo (2014). Development of Autonomous Car—Part I: Distributed System Architecture and Development Process. *IEEE Transactions on Industrial Electronics 61*(12), 7131–7140.

Kain, T., J.-S. Müller, P. Mundhenk, H. Tompits, M. Wesche, and H. Decke (2020). Towards a Reliable and Context-Based System Architecture for Autonomous Vehicles. In *Proceedings of the 2nd Workshop on Autonomous Systems Design* (*ASD 2020*), pp. 1–6.

Kain, T., H. Tompits, J.-S. Müller, P. Mundhenk, M. Wesche, and H. Decke (2020). FDIRO: A General Approach for a Fail-Operational System Design. In *Proceedings of the 30th European Safety and Reliability Conference* (*ESREL 2020*).

Li, B., J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong (2009). EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing* (*CLOUD-II 2009*), pp. 17–24.

Maurer, M., J. C. Gerdes, B. Lenz, and H. Winner (2016). *Autonomous Driving: Technical, Legal and Social Aspects*. Springer.

Morrow, W. R., J. B. Greenblatt, A. Sturges, S. Saxena, A. Gopal, D. Millstein, N. Shah, and E. A. Gilmore (2014). Key Factors Influencing Autonomous Vehicles' Energy and Environmental Outcome. In *Road Vehicle Automation*, pp. 127–135. Springer.

Pourmohseni, B., S. Wildermann, M. Glaundefined, and J. Teich (2017). Predictable Run-Time Mapping Reconfiguration for Real-Time Applications on Many-Core Systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems* (*RTNS 17*, pp. 148–157.

SAE International (2018). Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. In *SAE Standard J3016*, pp. 1–16.

Schrijver, A. (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc.

Shachnai, H. and T. Tamir (2001). On Two Class-Constrained Versions of the Multiple Knapsack Problem. *Algorithmica 29*(3), 442–467.

Tang, C., M. Steinder, M. Spreitzer, and G. Pacifici (2007). A Scalable Application Placement Controller for Enterprise Data Centers. In *Proceedings of the 16th International Conference on World Wide Web* (*WWW 2007*), pp. 331–340.

Weiss, P., A. Weichslgartner, F. Reimann, and S. Steinhorst (2020). Fail-Operational Automotive Software Design Using Agent-Based Graceful Degradation. In *Proceedings of the Conference on Design, Automation and Test in Europe* (*DATE 2020*).