This is the peer reviewd version of the followng article:

Adaptive Localization for Autonomous Racing Vehicles with Resource-Constrained Embedded Platforms / Gavioli, F., Brilli, G., Burgio, P., Bertozzi, D.. - 13:(2024), pp. 1-6. (2024 Design, Automation and Test in Europe Conference and Exhibition, DATE 2024 Valencia 25-27 Marzo 2024) [10.23919/date58400.2024.10546748].

Institute of Electrical and Electronics Engineers Inc.
Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

10/10/2026 13:09

(Article begins on next page)

Adaptive localization for autonomous racing vehicles with resource-constrained embedded platforms

Federico Gavioli*, Gianluca Brilli*, Paolo Burgio*, Davide Bertozzi†
* *University of Modena and Reggio Emilia, Italy, 224833@studenti.unimore.it, {gianluca.brilli, paolo.burgio}@unimore.it*
† *University of Manchester, United Kingdom, davide.bertozzi@manchester.ac.uk*

***Abstract*—Modern autonomous vehicles have to cope with the consolidation of multiple critical software modules processing huge amounts of real-time data on power- and resource-constrained embedded MPSoCs. In such a highly-congested and dynamic scenario, it is extremely complex to ensure that all components meet their quality-of-service requirements (e.g., sensor frequencies, accuracy, responsiveness, reliability) under all possible working conditions and within tight power budgets. One promising solution consists of taking advantage of complementary resource usage patterns of software components by implementing dynamic resource provisioning. A key enabler of this paradigm consists of augmenting applications with dynamic reconfiguration capability, thus adaptively modulating quality-of-service based on resource availability or proactively demanding resources based just on the complexity of the input at hand. The goal of this paper is to explore the feasibility of such a dynamic model of computation for the critical localization function of self-driving vehicles, so that it can burden on system resources just for what is needed at any point in time or gracefully degrade accuracy in case of resource shortage. We validate our approach in a harsh scenario, by implementing it in the localization module of an autonomous racing vehicle. Experiments show that we can adapt to variations in operational conditions such as the system workload, and that we can also achieve an overall reduction of platform utilization and power consumption for this computation-greedy software module by up to $1.6\times$ and $1.5\times$, respectively, for roughly the same quality of service.**

I. Introduction

Embedded processing systems for Autonomous Vehicles (AVs) must timely process a significant amount of information in order to interpret the external world and localize the vehicle, ultimately enabling a safe and efficient driving experience [1]. System engineers are looking with interest at embedded and power-efficient Multi-Processor Systems-on-Chip (MPSoCs), featuring few coarse-grained computing cores and several co-processors such as integrated GPGPUs, reconfigurable logic or ASIC accelerators [2]–[4] . Despite being less powerful than "traditional" multi- and many-cores from the High-Performance Computing (HPC) domain [5], these platforms outperform them in terms of power efficiency, smaller size and lower hardware complexity, making them the preferred choice for future self-driving vehicles of practical interest.

Real software stacks for self-driving vehicles typically consist of several computation-greedy software modules that end up being consolidated onto the same hardware platform to reduce the complexity of the networked system [6]. As a result, they end up competing for the same constrained set of computational resources (e.g., less than 10 general purpose cores), which makes it challenging for them to keep up with input data volumes and timing constraints, as well as to preserve their quality metrics over time. This situation is even exacerbated in real-life vehicles, where critical application must also co-exist with non-safety critical modules, such as telemetry, HVAC and infotainment, and where the working conditions (the so-called *Operational Design Domain – ODD [7]*) are extremely dynamic and unpredictable. This problem easily becomes a showstopper, and calls for more efficient strategies to unlock the full potential of the parallel hardware through efficient sharing.

To this extent, this paper advocates that one of the most promising techniques for efficient resource sharing consists of taking advantage of complementary resource usage patterns of software components through dynamic resource provisioning [8]–[11]. A key enabler of this paradigm consists of augmenting applications with dynamic reconfiguration capability, thus adaptively modulating quality-of-service based on resource availability or proactively demanding resources based on the complexity of the input at hand.

Exposing such a dynamic reconfigurability would open up new opportunities for system optimization and for operating optimally in a wide range of situations (especially conditions that threaten successful execution). On the one hand, applications would request resources adaptively by following a "use-just-what-you-need" philosophy, which avoids the inefficient sizing of resource quotas based on worst-case requirements. On the other hand, critical applications would be able to gracefully degrade their quality-of-service (yet still fulfilling their mission) in case of resource shortage.

To validate our intuition, we deploy on an representative embedded multi-core computer a highly-demanding, safety-critical application from the challenging scenario of autonomous racing vehicles. In particular, autonomous racing exacerbates the criticality and the requirements of the localization function, since when driving near the limits of handling the system has minimal margin for errors. Thus, the quality metrics of localization algorithms turn out to be highly sensitive to the interaction with other aggregated software modules.

For this reason, the ultimate goal of this paper is to explore the feasibility of a dynamically-reconfigurable application model for the critical localization function of self-driving vehicles, so that the latter can safely navigate the environment while dynamically adapting its configuration to changing internal and external operating conditions. In particular, we focus on Particle Filter (PF) [11], [12], a mainstream Montecarlo-based algorithm that locates the ego-vehicle within a given map, which is state-of-the-art for 2-D based LiDAR localization. Historically, researchers proved [13], [14] that it is possible – and effective – to dynamically adjust the operational parameters of this algorithm, with the goal of maximizing its accuracy. However, they do not consider context nor platform information to tune the dynamic adaptivity.

This work improves upon state-of-the-art in a twofold direction. On the one hand, we explore the dynamic reconfigurability of the PF in a multi-dimensional optimization space, spanning number of particles, number of cores, response latency and accuracy. On the other hand, we set up a methodology for characterizing the correlation among the above parameters for

the different sections of the racing circuits. Then, we deploy a semi-static reconfiguration strategy that statically computes the best PF configuration for a specific resource budget and circuit section, and dynamically switches the operating condition as the racing car moves across the map. We compare traditional, fully-static configuration strategies against our semi-static approach. We spot critical operating conditions associated with system overloading and harsh circuit sections where traditional approaches largely fail to meet real-time constraints (in our case, the frequency of the LiDAR sensor, i.e., 40Hz). In contrast, our approach is able to meet them by gracefully degrading the localization error while still keeping it within acceptable bounds. Moreover, using the same technique, we could proactively reduce the number of resources (threads/-cores) requested by the application while still meeting its timing constraints for roughly the same quality of service. In this case, our experiments show an overall reduction of platform utilization and power consumption by $1.6\times$ and $1.5\times$, respectively.

II. Related works

A. Dynamic allocation of resources

In this field, a wide literature spawns across the last decades, from the moment when multi-threading and multi-process technologies became popular. For reasons of space, we report the works that are closest to our approach, either because they target similar hardware, or because they are from the same application domains.

Koduri et al. [8] propose an architecture that groups cores, and allocate them to applications, in a many-core systems. Unlike us, they target NoC-based many-cores (i.e., 16+ cores), and offer a generic solution for any combination of architectures. Moreover, we have a totally different approach, because we suggest to include *qualitative* application-specific performance indicators, such as the accuracy, in our solution.

Shamsa et al. [9] propose a resource management system targeting heterogeneous multi-core platforms. The aim of the proposed framework is to find an optimal resource allocation for concurrent applications while optimizing performance and energy consumption. The authors proposed an evaluation using a Particle Filter algorithm, but unlike our work, they didn't provide an evaluation on a real use-case nor analyzed the Particle Filter accuracy.

There is a literature on Work-Stealing, such as the works from Cheng [15] and Marongiu [16]. State-of-the-art in the field targets architectures that greatly benefits by the adoption of explicitly managed ScratchPad Memories (SPMs). Our approach does not make any level of assumption on any underlying hardware, and, moreover, we include application specific metrics and parameters (i.e., particles) as "tuning knobs", in our scheduler.

From the field of robotics, Farinelli and Iskandar [17], [18] proposed online techniques to dynamically assigning threads to running applications. Both works are quite different compared to our work. The first one is focused on a multi-robot patrolling application, where the authors evaluated their *DTA-Greedy* and *DTAP* algorithms. The aim of the latter is to dynamically assigning threads to reduce the energy consumption of applications while still preserving the requirements. Unlike our work, their contribution is generic and it is not focused on a real-time localization as ours.

B. Adaptive Particle Filters

Literature offers a number of works with algorithmic solutions to reduce the computational complexity, or platform-specific accelerators, to Particle Filtering localization on constrained embedded devices for robots or cars. For instance, Goksoy [10] proposes what he calls "dynamic adaptive scheduling (DAS)". The main focus of this work is to optimize the task execution time and the scheduling overhead. However, unlike our work, application-specific parameters (like number of particles) are not considered.

Krishna [19], Chau [20] and Bernardi [11] target hardware accelerators in an FPGA-based systems. These approaches employ fully static design space exploration loop to find the optimal trade off between execution time and accuracy. Interestingly, the latter [11] shows how the number of particles could be adapted (in a fully static manner, in their case) to meet application timing constraints, which gives support to our intuition.

Fox [21] proposes a particle resampling method based on the Kullback-Leibler Divergence (KLD) metric to dynamically adapt the size of the particle set based on an estimate of the approximation error. Charroud [22] proposed a method to keep only relevant particles, using k-means and sigma points algorithms, with the aim to decrease the computational complexity of the Particle Filter, applied to real-time systems. Other works, proposed by Nagavenkat [23] are an hybrid approach based on a global/local scan matching to keep only relevant particles on the *Resampling* stage of the Particle Filter. Similar to the previous one, Kümmerle [24] reduced the computational complexity of the Particle Filter exploiting lightweight and compact geometric primitives.

As shown, all of these works are part of the wide(st) known literature on adaptive Monte-Carlo methods, and on Particle Filters more specifically [13], [14], [25]. However, in all of them, the adaptive behavior has the goal of minimizing the localization error, and doesn't consider any other system bounds like latency or core utilization. Furthermore, none of them explores how the PF component might co-exist with other critical applications on resource-constrained embedded platforms.

III. System design

This section describes the target system. We first introduce the target AV platform, namely a 1:10 race car called F1/10 [26], then we discuss the Particle Filter algorithm. Finally, we show our semi-static thread adjustment implementation, which we validate in the next section.

A. Target system

The F1/10 racing vehicle is a scaled prototype platform enabling research and teaching in the field of AVs. The vehicle is built to be representative of a real vehicle. Figure 1 shows the sensor set of the vehicle, and a typical AV stack. It is based on ROS2 [6], a typical choice in modern robots. This prototype is equipped by a 2D LiDAR sensor that produces a *point-cloud*, which directly feeds the PF algorithm, in conjunction with the wheel odometry data and – a typical choice – an offline pre-computed map. Our specific implementation of the Particle Filter will be explained in depth in the following subsection.

After the localization step, there comes the trajectory planning and control components, for which we have two possible options. The first is a non-reactive Pure Pursuit (PP) [27]

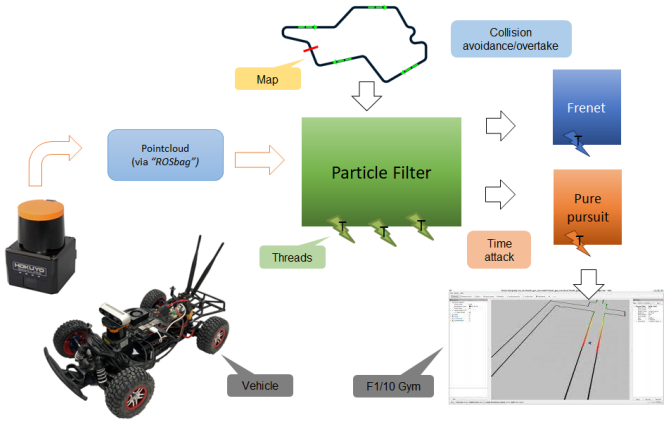

Fig. 1: Target AV stack.

controller, typically used for time attacks, such as racing qualifications. It assumes that an offline pre-computed optimal trajectory for the track is known, which is quite common in most of the races. This approach, however does not consider any obstacle intersecting the optimal trajectory. On the other side, PP can also be integrated, or replaced, with a reactive *local* approach, e.g., to generate multiple trajectories on-the-fly, in case the ideal pre-computed one is occluded, or if we are planning an overtake. We employ a state-of-the-art dynamic planner which features Frenet-frame trajectory generation [28]. In our experiments, both PP and Frenet act as concurrent applications that compete with PF for the available computing cores. Note that, they all are safety-critical components, hence, they all are at the same level of criticality/priority, in our system. In order to safely test and develop, the F1/10 platform comes with a simulator, called F1tenth Gym [29], which enables Hardware-in-the-Loop, and that we use in our experiments. Thanks to the adoption of the ROS2 framework, in conjunction with F1tenth Gym, we are able to generate race simulation recordings with synthetic sensor data inside real racetrack maps.

B. Particle filtering for AV localization

Formally speaking, Particle Filters solve a family of problems whose goal is to estimate the posterior probability distribution of the hidden model states: in our case, the "state" is the position of the vehicle in a known map. This particular technique, approximates the distributions of interest by means of random (weighted) samples, called *particles*. Applying this method to vehicle localization with 2D LiDARs, the weighted particle set represents an approximation of the probability distribution with relation to the real vehicle pose. The pose estimation process is based on the generation of a particle set, and each particle constitutes an *hypothesis* for the pose. Each hypothesis is then validated against the real world observation of the LiDAR sensor, and weighted.

In the first iteration, the algorithm is given a pose estimate and the particles are spawned with a uniform distribution around it. Each particle is represented as a 2D point with orientation (`x, y, yaw`). After this first initialization step, the following loop is executed to integrate each LiDAR and vehicle odometry measurement:

- *Motion model* – $\approx 5\%$ *exec time*: by leveraging a kinematic vehicle model, each particle is marched forward in time by integrating the vehicle wheel odometry data.
- Point Cloud *Downsampling* – $\approx 15\%$: Since the raw LiDAR data contains redundant information and is generally way too demanding to be processed on embedded processors, we downsample the point cloud from 1080 to 60 points. This step reduces the overall complexity of the subsequent ray marching phase.
- *Ray Marching* – $\approx 65\%$: For each particle, a set of rays is generated to simulate the LiDAR measurement of each pose candidate. This is done by marching a point in space until it reaches a wall in the racetrack map.
- *Weight Computation* – $\approx 15\%$: In this phase each particle in our hypothesis space is assigned a *weight* (i.e., a score) depending on a distance function and the measurement of the sensor. The expected value of the particle distribution is output as the pose estimate.
- After computing the expected pose of the distribution we *Resample* the particle set with a low variance approach.

Intuitively, spawning more particles create a more dense distribution, which leads to a more accurate localization. This, however requires more computational power and subsequently increases the overall response time. Nicely, particles can be processed independently, hence performance and accuracy scale increasing the number of parallel threads. We build on the most famous sequential implementation of the algorithm, namely, the one from MIT [12], and we provide a parallel version [1], that spawn hundreds-to-thousands of particles using OpenMP [30].

The most time consuming part of the application is the ray marching, and that's the best candidate for multi-threaded parallelization. Our goal is to determine/trade the optimal number of threads and number of particles under different system workloads in an adaptive manner.

C. Adaptive algorithm design and methodology

To provide a deterministic test infrastructure for the particle filter, we recorded a synthetic sensor dataset inside of the F1tenth Gym simulation environment [29]. We set up a Hardware-in-the-Loop (HiL) infrastructure with a desktop PC which simulates the sensors and an embedded board as the target hardware to run the localization software. As embedded platform, we used an NVIDIA Jetson, a typical choice for resource constrained AVs. The Jetson is equipped with a quad-core processor without superscalar extensions. Our localization software does not exploit the embedded GPU, we leave this exploration for future works. With HiL, we can collect high fidelity data regarding, e.g., the CPU usage, power consumption and localization latency of the algorithm, without requiring a real vehicle.

With this system in place, we statically characterize the workload and behavior of the PF application while varying the number of particles and degree of parallelism. We further split this data into different map sectors, to better analyze the changing demands of resources with relation to the morphology of the track (see Figure 2). Figures 3, 4 and Table I show the results of the characterization. Figures 3a and 3b show the execution time varying the number of particles (between 25 up to 1000) and threads for sectors 2 an 3 respectively [2]. The red planes represent the performance requirement, that is fixed at 25ms, and it is given by the frequency of the LiDAR, i.e., 40Hz. Intuitively, all the configurations that fall below the red plane are feasible configurations, while on the contrary,

[1] https://github.com/HiPeRT/particle_filter_dta

[2] For the sake of readability these results are presented as surface graphs, and for reasons of space we only plot the two most relevant sectors, but the others exhibit similar behavior.

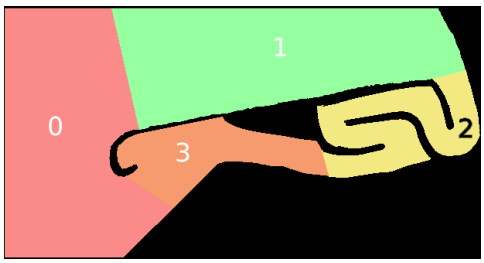

Fig. 2: The test racetrack, split in four sectors

# of Particles	Sector 2		Sector 3	
	RMSE [m]	Std. Dev. [m]	RMSE [m]	Std. Dev. [m]
25	0.4555	1.1086	1.8875	3.3803
50	0.3936	1.0012	1.5934	2.8955
75	0.1672	0.0105	0.9236	1.1903
100	0.1688	0.0111	0.6804	0.7533
200	0.1658	0.0090	0.4424	0.1788
300	0.1622	0.0065	0.3792	0.1655
400	0.1616	0.0066	0.3603	0.1615
500	0.1625	0.0063	0.3606	0.1482
600	0.1543	0.0055	0.3358	0.1193
700	0.1501	0.0046	0.2910	0.0773
800	0.1516	0.0045	0.3027	0.1051
900	0.1518	0.0041	0.2897	0.0753
1000	0.1509	0.0053	0.2606	0.0697

TABLE I: Localization error (RMSE and RMSE standard deviation) with varying number of particles.

all the points that are above the threshold don't respect the performance constraint and must be discarded.

An optimal configuration of particles/threads would prefer a higher number of particles, to achieve a better localization accuracy and a possibly low number of threads to proactively freeing computational resources to other applications on the autonomous driving stack.

Given the Particle Filter implementation discussed in Section III, we notice a directly proportional relation between the Ray Marching step execution time and the average wall distance. Following this rationale, sector 2 has a significantly lower computational load which also influences latency and power consumption.

Figure 4 report the CPU core utilization in percentage (Figures 4a and 4b), between 0% up to 400%, where 400% means that all the four cores are at full throttle. Figures 4c and 4d show the CPU power consumption, expressed in Watts. Intuitively more particles and threads would imply an higher CPU utilization, which in turn leads to a higher power consumption.

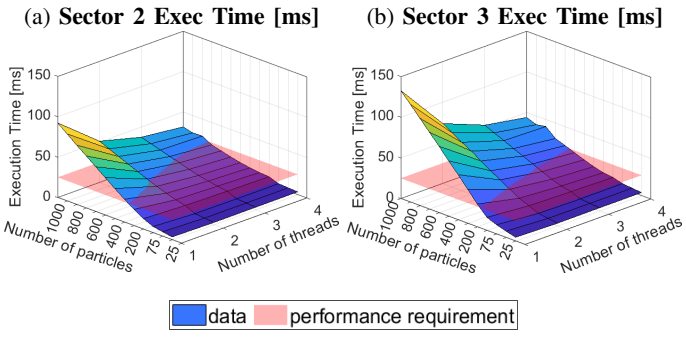

Fig. 3: PF execution time varying the number of particles and threads on sectors 2 and 3 of the race track. The red plane represents the performance requirement of 25ms.

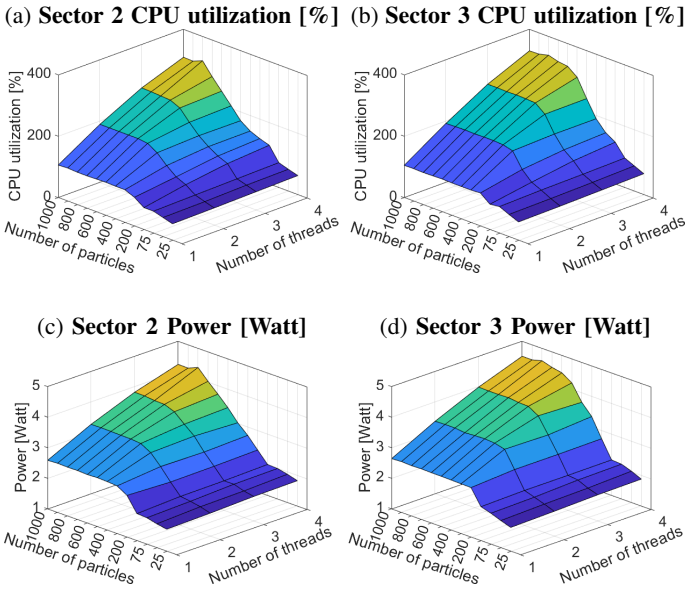

Fig. 4: CPU utilization and power consumption varying the number of particles and threads on **sector 2** and **sector 3** of the race track.

Different sectors may have different particle count requirements to avoid divergence and maintain the localization error low. This happens for example in sector 3, where high speeds and sudden steering movements are inaccurately portrayed by the motion model. This is noticeable by looking at the accuracy data in Table I.

Since particle filtering is a probabilistic approach, different test runs on the same sensor dataset may lead to different results. To characterize this variability, we run each configuration over a high number of test runs and evaluate localization accuracy using two metrics: the average RMSE (Root-Mean-Square Error) with relation to the ground truth trajectory, provided by simulator, and the RMSE standard deviation, which models the error variability during multiple test runs. Table I reports the accuracy of the Particle Filter localization measured as RMSE and its standard deviation. It's easy to figure out that few particles would imply higher localization errors with an high standard deviation. For example, considering the sector 2, we have ≈ 0.45m and ≈ 1.10m as RMSE and standard deviation, while these values go down to $\approx$ 0.15m and $\approx$ 0.005m maximizing the number of particles.

In the common static configuration case a certain static number of particles is chosen and refined for the whole racetrack. This fully static approach, however generates a configuration that is not always optimal in each sector. Our solution has the objective to adapt the behavior of the application (by adjusting the size of the particle set and the number of parallel threads) based on this offline characterization of the racetrack. At the end of each LiDAR integration loop, before the re sampling phase, we perform the live parameter adjustment. Pseudo-code is shown in Figure 5.

Initially, the application reads the available number of cores using the standard CPU affinity Linux kernel API. The available number of cores is interpreted as a core budget for the application. This core budget, in conjunction with the latency characterization data, is used to infer the optimal particle set size. For each sector, the optimal configuration takes into

```
sector_id     = get_sector_id (latest_pose);
num_cores     = get_core_budget ();
num_particles = get_optimal_config (sector_id, num_cores);
resample (num_particles);

// ...

#pragma omp parallel for num_threads (num_cores)
for (int p = 0; p < num_particles; p++)
    ray_marching (p);
```

Fig. 5: Pseudo-code that implements our approach

account i) the maximum number of threads, which is our CPU core budget, and ii) given the sector ID and the budget, the highest number of particles which guarantees the compliance with the response time bound. Once the thread count and particle set size are computed, we proceed with the resampling phase normally.

We also implement a *proactive* core release approach in sectors where the localization is "easy to compute". We decide to lower our core utilization in order to "make space" for other tasks, e.g., at low priority. To implement this, we reduce our CPU core budget, and re-compute the optimal number of particles using the same methodology we previously explained.

The use of offline characterization results makes our approach semi-static. However, we can easily implement a fully dynamic approach, where the same profiling gets updated at every lap in order to react dynamically to situations which are not covered by the offline characterization.

IV. Experimental validation

A. Setup

The target platform is a NVIDIA Jetson Nano [2], a low-end multi-core system that is already in production for AV systems, and it is perfect for deployment on the F1/10 car. It hosts a Quad-core ARM A57 1.43 GHz and 4GB of 64-bit LPDDR4 memory. It features no superscalar extensions such as hyperthreading, hence, we can safely assume that these four cores can sustain the computational bandwidth of exactly four software threads. The board also embeds an integrated GPU of the Maxwell family, with 128 CUDA cores, which we anyway don't consider in our scenario, because the CUDA abstraction layer is quite rigid, and does not provide much flexibility to dynamic reallocation of threads from the application level. Moreover, we plan to provide a methodology that is not specific on the given hardware provider. Since the target board runs a GNU/Linux Ubuntu distribution, with the standard GCC toolchain and OpenMP runtime support, it enables the widest applicability of our techniques to other platforms. Using this software stack, we can gather live information on the actual workload of the system, e.g., the utilization of ARM cores, or power consumption.

B. Methodology

As previously explained, we run the parallel Particle Filter that dynamically adjusts thread count and particle set size with the Frenet planner and the PP controller as single-thread interfering applications alongside it. We don't investigate the varying degree of parallelism and configuration space exploration of the interfering applications despite their capability to run multi-threaded because, in this work, we are not focusing on system level effects, hence we perform analysis and characterization of the PF only. The same characterization and adjusting methodology employed by the PF can also be replicated to any other parallel application in the driving stack. *System-level composability* of adaptive applications is a quite interesting topic, and deserves dedicated future research.

Sector	#Particles	#Threads	Latency [ms]	Utilization [%]	Power [mWatt]
S_0 (only PF)	400	4	24.81	2.8514	3896.0284
S_1 (only PF)	400	4	18.34	2.5273	3743.8056
S_2 (only PF)	400	4	13.39	1.7105	3021.9390
S_3 (only PF)	400	4	17.90	2.4301	3693.1250
S_0 (w/interf)	400	4	69.08	0.9419	2279.8286
S_1 (w/interf)	400	4	55.76	0.8511	2147.5556
S_2 (w/interf)	400	4	37.18	0.7189	1888.5048
S_3 (w/interf)	400	4	55.47	0.8315	2115.0000

TABLE II: Baseline for 400 particles and 4 threads, with and without interfering applications.

Sector	#Particles	#Threads	Latency [ms]	Utilization [%]	Power [mWatt]
S_0	400	4	24.81	2.8514	3896.0284
S_1	500	4	23.56	3.0899	4136.9874
S_2	700	4	23.90	2.8929	3926.6198
S_3	100	1	17.24	2.0194	3283.0331

TABLE III: **Scenario I**. When entering sector #3, the Particle Filter budget gets decreased to 1. In order to meet the latency constraints, the particle set size is reduced.

Let us now define a reasonable baseline with a static configuration of 400 particles and 4 threads. This is a potentially race-ready configuration that satisfies both latency and localization accuracy bounds in all four sectors. We will now compare this statically configured baseline (see Table II) with our adaptive approach in two different scenarios.

Scenario I: self-adaptive PF. We assume the Particle Filter components fully leverage the four available cores, each handling four threads processing 400 particles. As indicated in Table II, in Sector 3, computation takes 17.90 ms. Subsequently, a new process is initiated, reducing the computational allocation for the PF to a single core. Static approaches, incapable of dynamically adjusting thread and particle numbers (and consequently, computational workload), result in a component latency of 55.47 ms. This is approximately $3\times$ slower and leads to exceeding the deadline by over 30 ms, compromising the overall performance and safety of the entire autonomous driving stack. As illustrated in Table III, our system allows the Particle Filter to adapt its budget and particle set size (to one core and 100 particles, respectively), meeting computational constraints while experiencing minimal degradation in localization quality (-9.21%, relative to the baseline without interference).

Scenario II: proactively releasing cores. In this test case, reported in Table IV, the Particle Filter proactively reduces its budget to 1 thread. This lowers the core utilization ($1.6\times$) and power consumption ($1.5\times$), with respect to Table II, while maintaining the localization response time below the timing and with negligible accuracy loss ($< 1\%$, relative to the baseline without interference).

In this experiment, our goal was to prove the overall power reduction. One can follow a different strategy, so that, during this low-budget period, a new low-frequency low-priority task (e.g. online race trajectory optimization, or telemetry) could be launched on the now available cores. This opens up future developments on the composition of different application supporting this adjustment dynamically, at run-time.

We want to stress the fact that, in both scenarios, the particle filter adjusts the number of particles to the optimal value, delivering the highest localization quality for the sector given its computational budget, while meeting the latency constraint.

V. Conclusions

In this work, we present a novel approach to a self-adaptive system for localization in autonomous driving cars. Our ap-

Sector	#Particles	#Threads	Latency [ms]	Utilization [%]	Power [mWatt]
S_0	400	4	24.70	2.8674	3939.3248
S_1	500	4	23.40	3.0906	4148.2766
S_2	200	1	20.98	1.0142	1989.8885
S_3	500	4	24.15	3.0048	4111.0471

TABLE IV: **Scenario II**. In sector 2, the Particle Filter proactively frees up three cores (`budget` = 1) while adjusting the particle set size in each sector to meet the latency constraint

proach enables higher flexibility, and better resources optimizations with respect to traditional approaches, that don't take into account both application reconfigurability (in our test case, the number of particles of a Particle Filter) and system utilization at the same time. Our solution enables higher promptness and reactiveness of the overall system to unexpected workload variations, enabling safer driving, but it can also be adopted in a proactive manner, to tune system utilization, for instance to reduce power consumption $\approx 1.5\times$, in our experiments.

Our approach is semi-static, because it relies on precomputed performance tables. The next natural steps are to explore a fully dynamic approach, where these performance/accuracy profiles are updated on-the-fly while system is running, and to analyze system composability of multiple adaptive applications through their holistic orchestration and tuning.

Moreover, even if our approach can be quickly implemented on other application components that exhibit "knobs" such as particles number adjustment, it would be crucial to integrate it with existing parallel programming models (the natural candidate, in this case, could be OpenMP), to prove its widest applicability in an application-independent manner.

VI. Acknowledgements

This work is supported by the AI4CSM project, funded from the ECSEL Joint Undertaking (JU) under grant agreement No 101007326. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Austria, Belgium, Czech Republic, Italy, Netherlands, Lithuania, Latvia, Norway.

References

[1] R. Hussain and S. Zeadally, "Autonomous cars: Research results, issues, and future challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1275–1313, 2019.

[2] *Nvidia Jetson Nano Developer Kit.*, https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[3] Xilinx inc., "Versal AI Core Series." [Online]. Available: https://www.xilinx.com/products/silicon-devices/acap/versal-ai-core.html

[4] Mobileye, "Mobileye an intel company." [Online]. Available: https://www.mobileye.com/

[5] F. Nobis, M. Geisslinger, M. Weber, J. Betz, and M. Lienkamp, "A deep learning-based radar and camera sensor fusion architecture for object detection," in *2019 Sensor Data Fusion: Trends, Solutions, Applications (SDF)*, 2019, pp. 1–7.

[6] C. Gómez-Huélamo, A. Diaz-Diaz, J. Araluce, M. E. Ortiz, R. Gutiérrez, F. Arango, A. Llamazares, and L. M. Bergasa, "How to build and validate a safe and reliable autonomous driving stack? a ros based software modular architecture baseline," in *2022 IEEE Intelligent Vehicles Symposium (IV)*, 2022, pp. 1282–1289.

[7] "Sae j3016. taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles," in *Technical Report*, 2016.

[8] J. S. Koduri and I. Anagnostopoulos, "SPA: Simple pool architecture for application resource allocation in many-core systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1364–1368.

[9] E. Shamsa, A. Kanduri, A. M. Rahmani, and P. Liljeberg, "Energy-Performance Co-Management of Mixed-Sensitivity Workloads on Heterogeneous Multi-core Systems," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 421–427.

[10] A. A. Goksoy, S. Hassan, A. Krishnakumar, R. Marculescu, A. Akoglu, and U. Y. Ogras, "Theoretical Validation and Hardware Implementation of Dynamic Adaptive Scheduling for Heterogeneous Systems on Chip," *Journal of Low Power Electronics and Applications*, vol. 13, no. 4, 2023. [Online]. Available: https://www.mdpi.com/2079-9268/13/4/56

[11] A. Bernardi, G. Brilli, A. Capotondi, A. Marongiu, and P. Burgio, "An FPGA Overlay for Efficient Real-Time Localization in 1/10th Scale Autonomous Vehicles," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 915–920.

[12] C. H. Walsh and S. Karaman, "CDDT: Fast Approximate 2D Ray Casting for Accelerated Localization," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[13] Stahl, Tim, Wischnewski, Alexander, Betz, Johannes, and Lienkamp, Markus, "Ros-based localization of a race vehicle at high-speed using lidar," *E3S Web Conf.*, vol. 95, p. 04002, 2019. [Online]. Available: https://doi.org/10.1051/e3sconf/20199504002

[14] V. Elvira, J. Miguez, and P. M. Djurić, "On the performance of particle filters with adaptive number of particles," *Statistics and Computing*, vol. 31, no. 6, p. 81, Oct 2021. [Online]. Available: https://doi.org/10.1007/s11222-021-10056-0

[15] L. Cheng, M. Ruttenberg, D. C. Jung, D. Richmond, M. Taylor, M. Oskin, and C. Batten, "Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 46–58. [Online]. Available: https://doi.org/10.1145/3582016.3582020

[16] A. Marongiu, P. Burgio, and L. Benini, "Vertical stealing: robust, locality-aware do-all workload distribution for 3D MPSoCs," in *CASES 2010*, V. Kathail, R. Tatge, and R. Barua, Eds. ACM, 2010, pp. 207–216. [Online]. Available: https://doi.org/10.1145/1878921.1878952

[17] A. Farinelli, L. Iocchi, and D. Nardi, "Distributed on-line dynamic task assignment for multi-robot patrolling," *Auton. Robots*, vol. 41, no. 6, pp. 1321–1345, 2017. [Online]. Available: https://doi.org/10.1007/s10514-016-9579-8

[18] V. Iskandar, C. Salama, and M. Taher, "Dynamic thread mapping for power-efficient many-core systems under performance constraints," *Microprocessors and Microsystems*, vol. 93, p. 104614, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933122001557

[19] A. Krishna, A. van Schaik, and C. S. Thakur, "FPGA Implementation of Particle Filters for Robotic Source Localization," *IEEE Access*, vol. 9, pp. 98 185–98 203, 2021.

[20] T. C. P. Chau, X. Niu, A. Eele, J. Maciejowski, P. Y. K. Cheung, and W. Luk, "Mapping Adaptive Particle Filters to Heterogeneous Reconfigurable Systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, dec 2015. [Online]. Available: https://doi.org/10.1145/2629469

[21] D. Fox, "KLD-Sampling: Adaptive Particle Filters," in *Advances in Neural Information Processing Systems*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds., vol. 14. MIT Press, 2001.

[22] A. Charroud, K. Moutaouakil, and A. Yahyaouy, "Fast and accurate localization and mapping method for self-driving vehicles based on a modified clustering particle filter," *Multimedia Tools and Applications*, vol. 82, 11 2022.

[23] N. Adurthi, "Scan matching-based particle filter for lidar-only localization," *Sensors*, vol. 23, no. 8, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/8/4010

[24] J. Kümmerle, M. Sons, F. Poggenhans, T. Kühner, M. Lauer, and C. Stiller, "Accurate and efficient self-localization on roads using basic geometric primitives," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 5965–5971.

[25] Sampo Kuuttiet al., "A survey of the state-of-the-art localisation techniques and their potentials for autonomous vehicle applications," *IEEE Internet of Things Journal*, vol. PP, pp. 1–1, 03 2018.

[26] O'Kelly, Matthew and Sukhil, Varundev and Abbas, Houssam and Harkins, Jack and Kao, Chris and Pant, Yash Vardhan and Mangharam, Rahul and Agarwal, Dipshil and Behl, Madhur and Burgio, Paolo and others, "F1/10: An open-source autonomous cyber-physical platform," *arXiv preprint arXiv:1901.08567*, 2019.

[27] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-92-01, January 1992.

[28] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a frenét frame," in *2010 IEEE International Conference on Robotics and Automation*, 2010.

[29] "The F1tenth Gym simulator," https://github.com/f1tenth/f1tenth_gym.

[30] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.