Model predictive control for electric vehicle charging using low power microcontrollers

Kevin Böhmer
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
K.Boehmer@student.utwente.nl

ABSTRACT

This research investigates the implementation and performance of model predictive control algorithms for electric vehicle charging on low power microcontrollers with limited resources. The ESP32, ESP32-S2, and ESP8266 boards are chosen as the test platforms, because the boards are cheap and have integrated onboard Wi-Fi. The chosen microcontrollers can be considered as an edge computing node that can provide resilience to the future (smart) power grid. This research investigates which adjustments are needed to run a model predictive scheduling algorithm on a microcontroller with sufficient performance. The performance will be compared with a similar algorithm that runs on a reference system. Such a smart electric vehicle control node can interact with other devices using Demand Side Management. Through performance tests, we demonstrate that the ESP32 is able to run the discrete scheduling algorithm with 1440 intervals in 761.7 milliseconds. The ESP32 needs 7.93 milliseconds to compute a schedule with the continuous scheduling algorithm for the same number of intervals.

Keywords

Model predictive control, microcontroller, Demand Side Management (DSM), Profile Steering, Smart Grid, Smart Charging

1. INTRODUCTION

Most of the current existing electricity networks were designed decades ago. These networks were not designed to cope with relatively high and potentially synchronized loads such as electrical vehicles and heat pumps. Additionally, the number of solar panel installations is increasing, resulting in increased decentralized energy generation. Moreover, the popularity of electric vehicles (EVs) is also rising. These developments result in a major impact on the electricity network because the battery capacities of electric vehicles contain much energy and their chargers require relatively high power. For example, the current Tesla Model S has a 100 kWh battery. In contrast, an average Dutch household uses 10 kWh per day. Research has shown that the electricity network cannot cope with such high power loads and energy [8].

35th Twente Student Conference on IT July 2nd, 2021, Enschede, The Netherlands.

All the above-mentioned factors lead to severe peaks in the consumption and production in the electricity network. Demand and supply should be in balance because cheap storage does not yet exist in an electricity network. Demand Side Management (DSM) approaches can flatten out these peaks [16]. Many of these DSM algorithms use energy prices as steering signals. Gerards et al. [5] presented the profile steering algorithm. This algorithm uses desired power profiles as steering signals instead of energy prices. Van der Klauw [16] provided a scheduling algorithm that can optimize device operations according to these desired power profiles. This paper also shows that steering signals based on energy prices may result in power quality problems and high losses in performance and that steering using profiles yields better results. According to a schedule created by an optimization algorithm, smart charging EVs can help prevent the power grid from overloading. The optimization algorithm requires information such as profile steering profile, arrival time, departure time, required charging, and maximal charge rate.

The research in this paper investigates if the proposed scheduling algorithm presented in [16] can be implemented on a low powered microcontroller, in which the computational resources are limited. The advantage of using microcontrollers for the control of EV chargers is mainly the inexpensive price of a microcontroller. This is initial research towards a cost-effective solution for a smart mesh network that could potentially control a fleet of EVs or remote microgrids in developing countries fully powered by renewable energy.

The goal of this research is to implement a model predictive algorithm for one EV charger that can be run on one energy efficient microcontroller.

To achieve this goal the following research questions (RQ) are defined:

- RQ1: How can a model predictive algorithm be implemented on a low powered microcontroller for one EV charger?
- RQ2: What are the limitations of a low powered microcontroller for running this algorithm?
- RQ3: What performance trade-offs can be made in the implementation and what is the impact of such trade-offs on the performance?
- RQ4: How does this algorithm on the controller compare to the current application on the reference system?

First, the necessary background information is explained. Section *2. Background* is followed by section *3. Algorithm*, which explains the used algorithms in detail. The requirements and systems will be explained in the section: *4. Test platform.* The next section is *5. Evaluation method*, which describes the tests that will be performed. The results of these tests can be found in the section *6. Results.* This

section is followed by the *7. Conclusion* and *8. Future work*.

2. BACKGROUND

Gerards et al. [5] performed research in energy prices and power profiles as steering signals. Their research shows that price steering may result in power quality problems and high losses in performance. They propose an algorithm for energy coordination using profiles instead of prices. In a test-case, this algorithm results in a much flatter power profile. It also reduces distribution losses by 48% compared to uniform pricing.

Van der Klauw [16] shows that many of the scheduling algorithms that use profile steering as steering signals fall into the class of resource allocation problems. Scheduling an EV charger is similar to a classical resource problem if it can charge at any rate between zero and a given maximum. If the EV can only charge at a particular number of rates, this problem becomes NP-hard. In practice, EVs can in general only charge at a set of fixed rates due to technical limitations of the EV and the charger. The research of Van der Klauw [16] gives an algorithm for solving this problem. Schoot Uiterkamp et al. [14] proposed an online scheduling algorithm for a single EV charger that does not require predictions of uncertain power consumption and production. With a characterization of the optimal solution, which can be easily found at the start of the planning, the optimal solution can be reconstructed relatively easily. The simulations show that this approach is robust against prediction errors in the characterizing value and this value can be calculated accurately using the historical data. He et al. [6] proposed an optimal scheduling algorithm for charging and discharging of EVs. Their solution consists of two parts: a globally optimal scheduling solution that can minimize the total cost and a local scheduling optimization. The globally scheduling scheme is impractical, because the arrivals of EVs in the future are unknown, the baseload in the future is not known and it is not scalable for a large number of EVs. Therefore a local scheduler is used which tries to achieve the performance of the globally scheduling algorithm closely. This paper does not mention the performance of the proposed scheduling algorithms and whether they can be implemented on microcontrollers.

Most research that has been performed with microcontrollers and smart charging only uses the microcontroller for communication. Martinenas et al. [12] researched smart charging using dynamic price signals. For charging the car, a Vehicle Smart Link (VSL) is used, this embedded computing device implements the IEC 61851 standard, which allows this device to function with a standard Electric Vehicle Supply Equipment (EVSE). This way the microcontroller can control the charging of the car. The other task of the VSL was to communicate with an external Smart Charging Controller with HyperText Transfer Protocol (HTTP). The external Smart Charging Controller performed the following tasks: monitoring the price stream and the EV status, calculating and updating the EV charging schedule, and dispatching control signals to the microcontroller connected to the EV.

Although there has been research done in DSM, microgrids, profile steering, and smart EV chargers, research that implements a smart EV scheduling algorithm on a microcontroller is currently lacking. This research contributes by implementing a model predictive algorithm for one EV charger that can be executed on a low powered microcontroller. We do this by investigating which adjustments are needed to implement the scheduling algorithm on a microcontroller and perform tests with measurements to validate the performance of this algorithm.

3. ALGORITHM

This section explains the working of the algorithms that are implemented in this research.

3.1 DSM with Profile Steering

Profile steering [16] is an iterative DSM algorithm. The goal of this DSM algorithm is to achieve a particular power profile by controlling devices, such that the aggregated power profiles of all devices approach that power profile closely. All connected devices try to minimize the Euclidean (2-norm) distance between their scheduled power consumption and the desired profile. These power profiles are lists of discrete time intervals. Each time interval represents the consumption or production over a fixed time. The smaller the time interval, the more accurate a certain desired power profile can be achieved. However, a larger amount of intervals will cause more computing power for the scheduling devices. The DSM algorithm runs on one computing device and communicates with a large number of devices. To realise this approach in practice, a tree structure of DSMs can be used in order to speed up the scheduling progress. One DSM computing node can be made responsible for only one neighbourhood, this DSM computing node communicates with a different DSM computing node that is responsible for multiple neighbourhoods. A DSM node sends the desired power profile to each connected device, an example of a tree structure can be seen in Figure 2. Every device optimizes its own power profile accordingly and sends back a new version of the power profile. The DSM node receives all the new power profiles and chooses the power profile which helps the aggregated power profile the most. The updated aggregated power profile is sent to all devices again and this process stops when there is no significant improvement anymore. See Figure 1 for an example of a 24 hours profile steering signal with an interval length of 5 minutes, resulting in 96 intervals. Figure 3 shows the output of the discrete and continuous EV charging algorithm with the DSM input from Figure 1

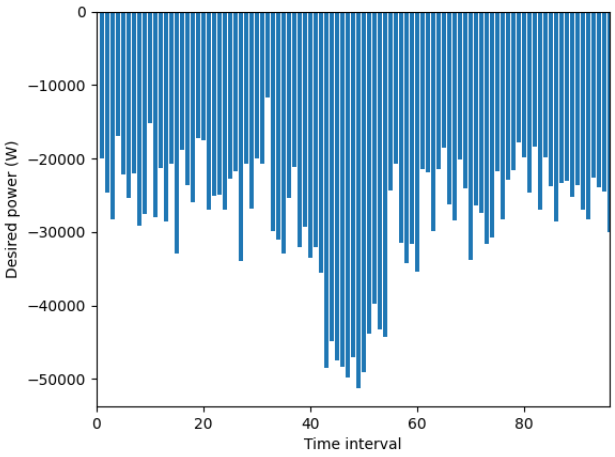

Figure 1. The 24 hours profile steer signal that is given to the scheduling algorithm, with 96 intervals

3.2 EV scheduling algorithm

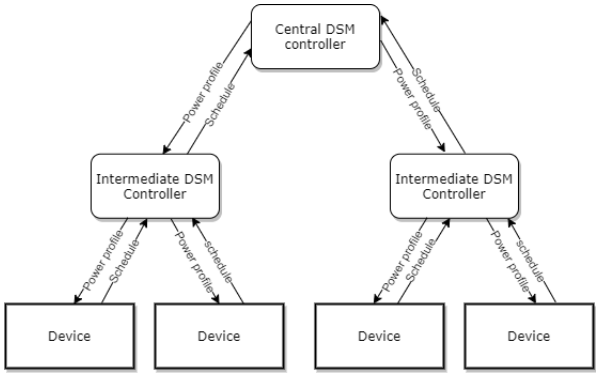

Figure 2. Example of a possible tree structure of DSMs with one intermediate layer

In this research, two different EV scheduling algorithms are implemented and tested, a continuous and a discrete variant [16]. The goal of the scheduling algorithm is to minimize the distance between the desired profile and the charging schedule. The discrete algorithm creates a schedule with discrete charging powers. These charging powers are predefined. The continuous algorithm produces a schedule with continuous charging powers. Due to the physical limitations of the EV and charger, EVs can only charge at fixed rates, therefore the discrete variant is more feasible to realise in practise. Nevertheless, because the charge rates are not fixed the scheduling problem is easier to solve, resulting in a lower computation time and hence used for comparison. A mathematical model is used for creating the charging schedules. The following inputs are used: capacity, type car, start time, end time, and the requested energy. Based on these inputs predictions have to be made, resulting in a model predictive control.

3.2.1 *Discrete EV planning algorithm*

This algorithm creates a planning to charge the car with discrete power values. Initially, for each time interval, a slope is calculated and stored in a list. Not only the slope value is stored, but also the *chargingPower*, which is used for calculating that slope value, and an integer storing the id of the time interval. This slope value is calculated with Equation 1. Initially, the *previousChargingPower* variable is zero and the *nextChargingPower* is the smallest non-zero charging power. The variable desired is equal to desired power for that particular time interval.

$$slope = ((nextChargingPower - desired)^2 - (previousChargingPower - desired)^2)/ (nextChargingPower - previousChargingPower) \quad (1)$$

When all slope values are calculated for each time interval, the list of slopes is sorted, the sorting algorithm is explained in Section 3.2.3. The first and therefore the smallest slope is stored in a temporary value and removed from the list of slopes. The *chargingPower* used for that slope value is subtracted from the remaining charge and added to the results. If there is still a higher *chargingPower* available, the next slope value is calculated using Equation 1. The difference between the initial phase is that the *previousChargingPower* is now a non-zero value. This slope value together with the *chargingPower* and the interval id is added to the list of slopes. This process repeats until the remaining charge is zero. The lowest slope value is chosen, because the smaller the slope value, the less difference between the charging schedule and the desired profile.

3.2.2 *Continuous EV planning algorithm*

This algorithm creates a schedule with continuous power values. This schedule is made with a piecewise linear objective function. For each interval a *lowerLevel* and *upperLevel* is computed, those values are stored in lists: *lowerLevels* and *upperLevels*. The *lowerLevel* is equal to the DSM power profile input for that particular time interval. The *upperLevel* is the sum of the DSM power profile for that particular time interval and the maximum charging power. Then two additional lists are created for storing the sorted results of the lists: *lowerLevels* and *upperLevels*. Then the theory of piecewise linear function is used to determine the breakpoint of the function [16]. Based on this breakpoint the charging strategy is determined.

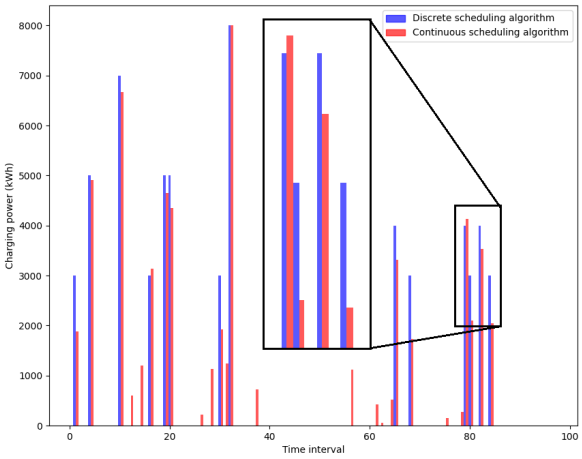

Figure 3. Output of the discrete and continuous EV charging algorithm

3.2.3 *Sorting*

Both the discrete and continuous algorithms make use of sorting. The function ***sort()*** is used for sorting the lists in the Python implementation. Python uses the Timsort algorithm for this function. Timsort has a worst-case performance of $\mathcal{O}(n \log n)$. And a best-case performance of: $\mathcal{O}(n)$. The ***qsort()*** function is chosen for the C implementation. The implementation of this sorting function depends on the C Standard Library (libc). For this research, the GNU project's libc implementation (GLIBC) is used. This ***qsort()*** is implemented as a merge sort algorithm and requires at least $\mathcal{O}(n)$ additional memory. If that memory is not available a fallback is made to the in-place sorting algorithm quicksort. Merge sort has the same worst-case and best-case performance as the Timsort algorithm. The Quicksort algorithm has a different worst-case performance: $\mathcal{O}(n^2)$

4. TEST PLATFORM

This section describes the requirements of the new system. Followed by suitable microcontrollers for the hardware platform. After this, the programming environment will be discussed of the system candidates. Lastly, the compiler settings are given.

4.1 Requirements

Pimentel [13] gives a structured insight into the field of design space exploration for embedded systems. It gives an overview of the basic concepts of design space exploration (DSE), which is a systematic analysis of multiple design points. For a DSE of embedded systems, multiple optimization objects should be considered simultaneously, for instance: performance, energy consumption, and cost. A DSE is used to define the requirements and selecting the target system. The system should have the following requirements:

- The most important requirement of the EV scheduling algorithm on the microprocessor is the performance of the scheduling algorithm. The profile steering algorithm is an iterative algorithm which means that multiple schedules need to be created by one device. All devices need to send an update of their schedule to the DSM if the schedule of another device gets accepted by the DSM. In this research we use 1 second as a limit to acceptable performance for use in profile steering.
- Another requirement of the system is to have support for Wi-Fi communication. This is used for sending the energy profile to the microcontroller and sending the energy consumption back to the DSM system. Although it is possible to introduce a second device that communicates between the microcontroller and the DSM system, it is cheaper to use one device.
- The SRAM memory should be large enough to store all the necessary data for the computations. The microcontroller needs to store multiple lists in memory, which will make extensive use of the available memory. The scheduling algorithm will only plan the upcoming 24 hours. That 24 hours will be split up into time intervals, the smallest time interval that will be used in this research is 60 seconds, yielding 1440 intervals. The planning algorithm makes use of multiple lists with the size of the number of intervals, for instance: the microcontroller needs to store the desired power profile in memory and it has to store a list with the results. Depending on the chosen algorithm some extra lists are needed for storing intermediate results. The C implementation of the continuous scheduling algorithm requires 28 bytes for each time interval. In the worst-case scenario with 1440 intervals, at least 40.320 KB is needed for array allocation. The discrete variant makes less use of the memory.
- The cost is also a requirement of the system. The main advantage of a microcontroller is the price. Because of the low price a higher and quicker deployment of smart EV chargers is possible.

4.2 Hardware platform

The ESP systems are low-cost Wi-Fi microcontrollers with a full TCP/IP stack. These systems are produced by the Chinese company Espressif Systems. Due to the integrated Wi-Fi, these systems are often used in many IoT projects. In this paper, three different devices from Espressif are used for running the algorithm and benchmarking: ESP8266, ESP32, and the ESP32S2. The ESP32 has replaced its predecessor the ESP8266. The specifications of the microcontroller are shown in the tables: Tables 1–3.

All these three devices are systems on chip (SoC) that use the 32-bit Xtensa computer architecture which is developed by Tensilica. The ESP32 is a dual-core system with two Harvard architecture Xtensa LX6 CPUs that can individually be controlled [1]. The Harvard architecture stores the machine instructions and data in separate memory units which are connected by different busses. Therefore these microprocessors are able to execute programs and access data independently, and therefore simultaneously. ARM machines, on the other hand, are von Neumann machines. This means that the code, hardware registers, and RAM can be accessed in one single address space. Therefore the application code can be executed from RAM, as opposed to Harvard machines where applications are run from flash. The chip has 520 KB on-chip SRAM for data and instructions. Some SoC (System on Chip) modules such as the Wrover, have 4MB of external ISP flash and an additional 8 MB of SPI PSRAM (Pseudo static RAM). The ESP32 also has a hardware 32-bit floating point unit (FPU), which will speed the floating points arithmetic [3]. The instruction set of this microcontroller also has support for zero-overhead looping. Zero-overhead looping is a feature of the instruction set allowing the hardware to repeat the body of a loop automatically, rather than requiring software instructions to keep track of the for loop. This will save CPU cycles and therefore time. The ESP32 has support for DSP instructions, like 32-bit multiplier, 32-bit divider, and 40-bit Multiple-accumulate operation (MAC). These DSP instructions are used for audio and video processing and will therefore not be used in this research.

The ESP32-S2 uses a single-core LX7 processor, which will be in general faster than the ESP32 with single-core applications [2]. Although, this new chip does not have a hardware floating point unit. It is expressed that this will have a large impact on the performance. The S2 variant also does not have the zero-overhead looping feature and misses the MAC operation in the DSP instruction.

The ESP8266 is the slowest and the oldest microcontroller, which has a single-core Tensilica L106 32 bit.

Table 1. ESP8266 specifications

CPU	Tensilica L106 32-bit
Max CPU clock speed	160 MHz
SRAM	160KB
ROM	0
Cache	32 KB instruction
Wifi	Wi-Fi 4 (up to 72.2Mbps)
Bluetooth	N/A
RTC memory	768B
ULP coprocessor	N/A

Table 2. ESP32 specifications

CPU	Tensilica Xtensa 32-bit LX6 dual-core
Max CPU clock speed	240 MHz
SRAM	520 KB
ROM	448 KB
Cache	64 KB
Wifi	Wi-Fi 4
Bluetooth	BLE 4.2 (upgrade to 5.0)
RTC memory	16 KB
ULP coprocessor	Yes

4.3 Programming environment

This section discusses the two different programming environments that are used in this paper. The work by Babi-

Table 3. ESP32-S2 specifications

CPU	Tensilica Xtensa 32-bit LX7
Max CPU clock speed	240MHz
SRAM	320KB
ROM	128KB
Cache	8/16KB (configurable)
Wifi	Wi-Fi 4
Bluetooth	N/A
RTC memory	16 KB
ULP coprocessor	ULP-RISC-V

uch, Foltýnek, and Smutný [1] shows how applications can be developed for the ESP32 microcontrollers family. This review gives an overview of the possibilities of application development for this platform in the area of data measuring and processing. It argues that the Arduino Core for ESP32 is the easiest solution for the development of applications. It is an open-source platform that is designed for Atmel microcontrollers, however, with plugins it is also possible to develop for other boards. Although, if the embedded system is more complex or needs more optimization, this research recommends using Espressif IoT Development Framework. With this tool it is possible to develop applications in a native way, programs can be created at the lowest level. The disadvantage of this approach is that the developer needs an initial high experience in order to develop an application. The third solution is to use MicroPython as the default firmware. MicroPython is a lightweight variant of Python, which includes some basic libraries of Python. MicroPython can be executed on microcontrollers. This development method is recommended for prototyping and will be used for testing the existing Python implementation [7].

In this research two different ways of programming these ESP boards are considered. Applications can be developed in the low-level programming language C, which will run on a Real-Time Operating System (RTOS). The official supported platform is ESP-IDF, this SDK can be used in Eclipse or Visual Studio Code. The compiled applications will run on the FreeRTOS operating system. Research [9] shows that this approach results in the most efficient instruction code, making the applications run the fastest, therefore this toolchain is chosen for the experiments. FreeRTOS allows easy multitasking between different tasks, which allows executing several tasks pseudo concurrently. This is very useful for applications that use for instance Wi-Fi, then the Wi-Fi code can run in a separate task and FreeRTOS will take care of the scheduling.

The second approach to program the ESP boards is by using MicroPython (MP). MP is largely compatible with the high-level programming language Python 3, it is written in C and is optimized to run on microcontrollers. MP is a full Python compiler and runtime that runs on the microcontroller's hardware, therefore applications can be run without being flashed. Writing code in a high-level programming language like Python is easier than in low-level programming languages like C. There are also some disadvantages of this approach, applications run much slower than compiled applications and uses more memory. The research of Ionsecu et al. [9] shows the performance differences between MicroPython and C for programming the ESP32 and STM32 microcontrollers. The research shows that the C implementation is faster in all different scenarios, but the difference varies on the task complexity and memory allocation. Different tests were performed in order to compare the performance differences between applications written in C and MicroPython on ESP32. One of the tests consists of a for-loop of length 300000 that performs an floating point addition operation. The C application was able to finish this test in 6257 μs. The MicroPython took a much longer time, namely 1021756 μs, this is a factor 163 difference. To avoid reloading the code into the MicroPython Environment on the microcontroller after a reset, frozen modules can be used [4]. A frozen module is a library that will be part of the firmware. This process requires a custom MicroPython build image. With this approach, the bytecode and the static data will be in flash during the execution. The RAM is only used for dynamic data. This will reduce the amount of RAM needed.

4.4 Compiler optimization

Compilers are the bridge between the written software and the machine code instructions that will be executed by the microcontroller [11]. The optimization process of the compiler is a non-mandatory phase, but can have a big influence on the performance. For the development of the C applications for the microcontroller the GCC compiler is used. The CNU Compiler Collection (GCC) is a GNU project which supports various programming languages. Without the optimization selected, the goal of the compiler is to reduce the cost of compilation and make debugging produce the expected results. Code statements are independent, allowing stopping the program with a breakpoint and reassigning new values to variables or change the program counter to any other statement and get exactly the right result that is expected from the source code. The optimization level can be set with the -On flag, where n stands for the specific optimization level. The Espressif toolchain has four default different compiling options:

1. Debug (-Og)
2. Optimize for size (-Os)
3. Optimize for performance (-O2)
4. Debug without optimization (-O0)

The flags refers to the GCC compiler flags, which are described in Table 4 [10].

Table 4. GCC optimizations

Flag	GCC optimization level
-O0	Default, no performance optimizations.
-O1 or -O	Decrease the code size with some increase in speed.
-O2	More speed optimizations as long as they do not increase in code size.
-O3	Most speed optimizations, but will lead in larger code size. Includes -O2 optimizations plus loop unrolling and inlining.
-Os	Only optimizations for code size, no speed optimizations are applied.

5. EVALUATION METHOD

This section describes the different test scenarios and how the tests are performed. Time measurement is performed in order to validate if the performance requirement can be achieved. With this time measurement, the impact of the interval size and possible limitations can be found in order to answer RQ3. This time measurement is also used for comparing the microcontroller implementation and the implementation on the reference system, this is used to

answer RQ4. The reference system consists of an i7-8750H 2.2 GHZ CPU and 16 GB RAM.

5.1 Test scenarios

This subsection explains the used test scenarios. Each algorithm is tested with two different desired profiles. The zero profile where the desired profile is a flat line with value zero. The other scenario is a curve with different values which represents a realistic power demand curve of a DSM with profile steering.

Both these two test scenarios are therefore tested with a different number of intervals: 96, 144, 288, 480, and 1440. These amount of intervals represent interval durations of: 15, 10, 5, 3, and 1 minute. The desired profile is stored in these intervals. A smaller interval duration results in larger profiles to cover the same time period resulting in more computations. For each interval duration, a predefined array is used, in this way the same test scenario can be tested on the microcontroller and the reference system. It is possible to use a seed to let have the random function the same output for every run, however these seeds are not interchangeable between the used programming languages: Python and C. The outcome of the random seed functions can even differ between different compilers. The hard-coded array for the 96 intervals, which represents the DSM Profile Steer input signal, can be seen in Figure 1.

For this test scenario, the scheduling algorithm charges one car with a capacity of 15 kWh. The discrete algorithm can charge the car with the following powers: 3000, 4000, 5000, 6000, 7000, and 8000 W. The maximum charging power for the continuous algorithm is also 8000 W. In order to measure the run duration of the algorithm more precisely, the total time for executing a thousand iterations is measured. An example of a discrete and continuous charging strategy for this test scenario can be seen in Figure 3. The section *6. Results* will show the execution time for one iteration.

5.2 Computation performance evaluation

Before the algorithms can be tested, the choice has to be made, which noninvasive or invasive way of performance measuring is used. The act of time measuring must not have a significant influence on the impact of the system [15]. For this research an invasive approach is chosen, the impact of measuring on the system is negligible, because the execution time of the algorithm is only printed when the measurements have been performed, so the ***printf()***, for displaying the results, does not influence the measurements. There are two different invasive methods that can be used. The first one is measuring with system call: ***clock()***. This system call returns an approximation of the processor time, by subtracting the clock after the algorithm with the clock before the start of the algorithm, the duration of the algorithm can be achieved. Another approach to measure the time is the FreeRTOS library call: ***xTaskGetTickCount()***. This call returns the number of ticks since the task was created. The default tick rate of FreeRTOS is 100 Hz. So 188 ticks is equal to 1.18 seconds. Table 5 shows that there is no difference between the measurements with these two system calls. These results also show that there is no performance difference between executing the algorithm in the main function or as a separate FreeRTOS task. Therefore we choose to run our experiments using the system call ***clock()*** and the algorithm is called from a separate FreeRTOS task.

5.3 Implementation and verification

The existing Python implementation was used for creating the new C implementation. The Python implementation was also used for verifying the correctness of the new C implementation. Different test scenarios, which can be found in the *Evaluation method* section, were executed on both implementations and the resulting profile is verified to be equal. Therefore, the C implementation is correct for at least the used test cases in this paper.

Table 5. Computation performance evaluation. A comparison between *clock()* and *xTaskGetTickCount()*.

	Task 1	Task 2	Task 3
ESP32 Task ticks (ticks)	118	123	19
ESP32 Task clocks (seconds)	1.18	1.23	0.19
ESP32 Main clocks (seconds)	1.18	1.23	0.19
ESP32-S2 Task ticks (ticks)	141	148	55
ESP32-S2 Task clocks (seconds)	1.41	1.48	0.55
ESP32-S2 Main clocks (seconds)	1.41	1.48	0.55

5.4 Compiler optimization evaluation

The four different compiler settings which are described in 4.4 are used for compiling a test case scenario in order to determine which compiler setting will be used for the performance results. The discrete and continuous algorithm with 140 intervals is tested on the ESP32. The execution time of both algorithms is measured as well as the usage of the DDRAM, IRAM and the total image size. The test results are displayed in Table 6. The *Optimize for size (-Os)* setting results in the smallest image size and IRAM usage. From these results, we can conclude that the performance setting (-O2) is about 6% faster for the discrete case and 9% faster for the continuous case. However, it is questionable to what extent these optimizations are representative for the test case. The DSM signal input is predefined as an array. Because this array is predefined the compiler has the possibility to optimize this array in the calculations. In the real-life scenario, this array would be received for instance via Wi-Fi and can never be optimized, because the data is variable. Therefore the choice has been made to use the *Debug (-Og)* setting for all the performance tests.

Table 6. Flags comparison

Flag	Discrete (ms)	Continuous (ms)	DDRAM (KB)	IRAM (KB)	Total (KB)
-Og	11.070	0.710	15	40	158
-Os	15.490	1.010	15	36	144
-O2	10.400	0.650	15	38	149
-O0	15.490	1.010	16	57	195

6. RESULTS

This section presents the results of the experiments.

6.1 Performance results

See Tables 8–11 for the test results, the execution times are for one iteration. The results of Table 8 are plotted in a logarithmic bar graph in Figure 4. The results of Table 9 are shown in Figure 5. Only the sample signals are plotted in a bar graph. The execution times of the microcontrollers for the discrete algorithm with a sample signal input are plotted in Figure 6.

6.1.1 Reference system

The C implementation of both algorithms on the reference system is in almost all cases faster than the Python implementation, except for the test instance with the discrete variant with a number of intervals of 1440 for the sample signal and the zero value signal. The difference between the C and Python implementation is larger for the continuous variant than the discrete variant. The C implementation is on average 7.1 times faster compared to the Python variant for the continuous test cases with a sample signal as input. The performance difference is even bigger when the continuous variant uses the zero signal as desired profile input. The C implementation finishes the algorithm 14.7 times faster than the discrete variant. The C variant is on average 1.2 times faster with the discrete test cases with a sample signal and 1.7 times faster when a zero value signal is used.

6.1.2 The microcontrollers

The ESP32 variant is in all test instances faster than the ESP32-S2 and ESP8266. The ESP32 is on average 2.0 times faster than the ESP32-S2 for the discrete variant with a sample signal as input. It is 1.5 times faster when the discrete algorithm uses a zero signal as input. The differences for the continuous variant are slightly larger, 2.5 for the continuous variant with a sample signal and 2.9 times faster for the continuous variant with a zero value signal as input. A possible explanation why the differences for the continuous variant are larger than the discrete variant is because the ESP32-S2 does not have an FPU, therefore all floating point arithmetic is implemented in software. A software implementation of floating point arithmetic results in more instruction code, this will take more time to compute. The ESP8266 is in all test cases slightly slower than the ESP32-S2. It is 1.2 times slower for the discrete variant and 1.3 for the continuous variant with the sample signal test case. The ESP8266 is with a zero signal on average 1.2 times slower with executing both algorithms.

6.1.3 Reference system versus ESP32

The reference system runs the discrete algorithm with a sample signal as input on average 30.6 times faster than the ESP32. The difference for the continuous variant is smaller, there is the difference of only a factor of 9.2. If a zero signal is used as input the discrete variant is in discrete case 20.8 times faster and 3.2 times faster for the continuous case.

6.2 Discrete algorithm with floats

This test compares the differences in computation time between integers and floats. Because calculations with floats require more clock cycles compared to calculations with integers, the execution time is higher. To see how much influence this has, the discrete algorithm is tested with two different flavors. The default integer flavor uses integers and the other flavor uses float values for the computations. Note that the output is still the same, the only difference is that the intermediate values are represented differently in memory and the arithmetic logic unit (ALU) will handle the calculations of both flavours differently. The results are shown in Table Table 7. From these results, we can conclude that the difference between floating points and integers operations in the discrete algorithm is negligible.

6.3 MicroPython

The MicroPython instance has been tested with an iteration size of 50 and an array size of 96 on the ESP32. The discrete variant with a sample signal as input takes 660 s and the continuous variant 20 s. Initially, more iterations were tested, but this resulted in an out-of-memory exception on the microcontroller. In contrast with C, Python makes use of garbage collection to free unused memory. In the C implementation, the same memory regions are used when calling the algorithm multiple times. The discrete variant is 1187 times slower than the native C implementation. The continuous variant is 350 times slower. With these results, we can conclude that for this application MicroPython is not a proper alternative for ESP-IDF toolchain.

Table 7. Test results of the discrete algorithm implemented with integers and floats for one iteration

Target device	Performance discrete algorithm with integers (ms)	Performance continuous algorithm with floats (ms)
ESP32	86.36	86.51
ESP32-S2	182.34	183.10

Table 8. Test results of the discrete scheduling algorithm with a sample signal as input for one iteration

# intervals	Ref. Py (ms)	Ref. C (ms)	ESP32 C (ms)	S2 C (ms)	ESP8266 C (ms)
96	0.1718	0.1205	5.49	11.12	12.61
144	0.2697	0.2462	11.06	22.35	28.18
288	1.3100	1.0315	30.13	61.16	74.53
480	3.0773	2.7584	86.35	182.35	208.28
1440	26.097	30.917	761.71	1599.2	1748.58

Table 9. Test results of the continuous scheduling algorithm with a sample signal as input for one iteration

# intervals	Ref. Py (ms)	Ref. C (ms)	ESP32 C (ms)	S2 C (ms)	ESP8266 C (ms)
96	0.0460	0.0055	0.46	1.14	1.48
144	0.0669	0.0091	0.71	1.75	3.17
288	0.1479	0.0204	1.24	3.08	3.93
480	0.3055	0.0397	2.21	5.63	6.63
1440	0.8118	0.1642	7.93	20.04	21.99

Table 10. Test results of the discrete scheduling algorithm with a zero signal as input for one iteration

# intervals	Ref. Py (ms)	Ref. C (ms)	ESP32 C (ms)	S2 C (ms)	ESP8266 C (ms)
96	0.2099	0.0852	3.64	5.19	6.17
144	0.3804	0.1790	7.28	10.56	14.05
288	1.1860	0.7802	25.45	37.85	43.13
480	2.9539	2.3953	66.78	100.25	114.11
1440	24.0058	23.586	562.92	858.14	1011.77

Table 11. Test results of the continuous scheduling algorithm with a zero signal as input for one iteration

# intervals	Ref. Py (ms)	Ref. C (ms)	ESP32 C (ms)	S2 C (ms)	ESP8266 C (ms)
96	0.07901	0.00458	0.25	0.75	0.82
144	0.11102	0.00770	0.38	1.12	1.23
288	0.24401	0.01574	0.75	2.21	3.79
480	0.37630	0.02900	1.26	3.67	4.04
1440	1.16764	0.08593	3.76	11.01	12.2

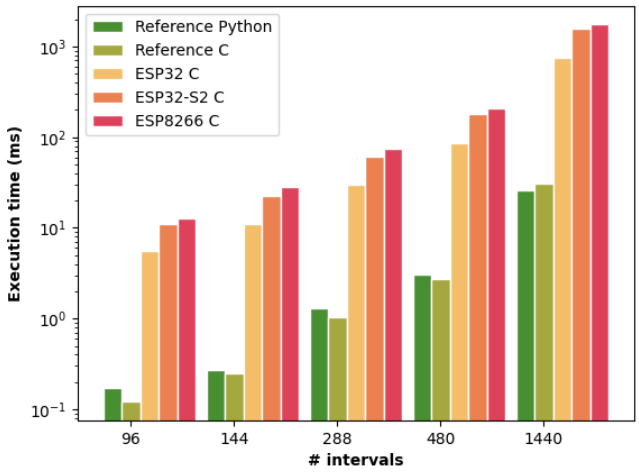

Figure 4. Test results of the discrete scheduling algorithm with a sample signal as input on a logarithmic scale for one iteration

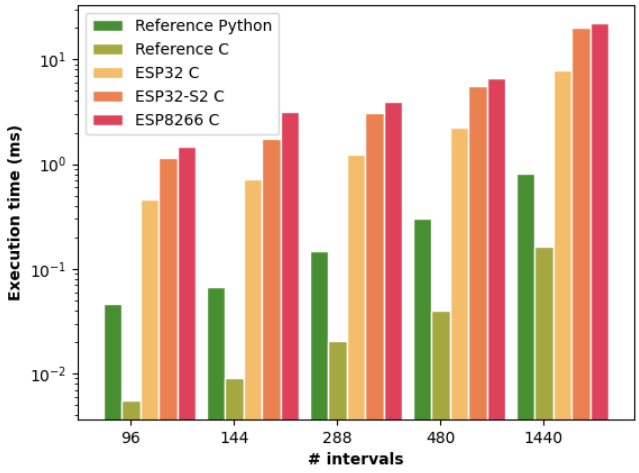

Figure 5. Test results of the continuous scheduling algorithm with a sample signal as input on a logarithmic scale for one iteration

7. CONCLUSIONS

The microcontroller is suitable for creating schedules for EVs with no problem as long as the number of intervals does not become too large. The ESP32 is able to create EV charging schedules in less than a second for every interval length. The ESP32-S2 variant can not satisfy this requirement only for the discrete variant with 1440 intervals, because it needs 1.6 seconds to create a schedule. The results show that the ESP32 is the fastest microcontroller in this research. The ESP32-S2, which has a newer instruction set LX7 compared to the LX6 of the ESP32, is slower in all test instances. The main cause is the lack of hardware implementation of floating point arithmetic. The C applications which run on FreeRTOS compiled by the ESP-IDF toolchain are much faster than the MicroPython instance. The main limitations of a microcontroller are the small amount of SRAM and the lower clock speeds. MicroPython is ideal for prototyping or small applications. However, it is not able to run large programs due to the low

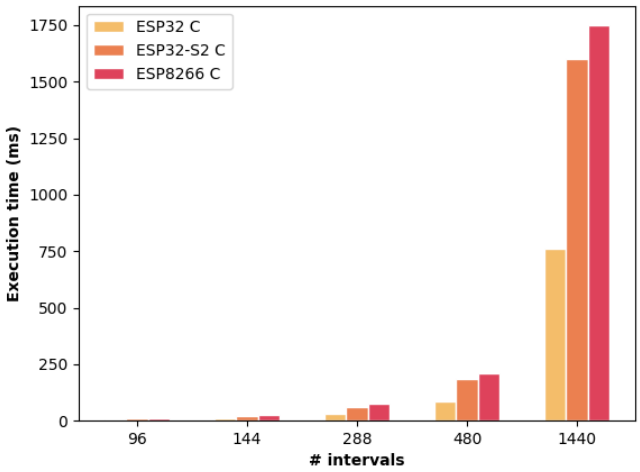

Figure 6. The microcontroller test results of the discrete scheduling algorithm with a sample signal as input for one iteration

availability of SRAM, therefore unsuitable for these applications. The ESP32 with MicroPython was only able to run the discrete variant with 96 intervals and 50 iterations. More intervals or a higher number of iterations resulted in memory problems. The main trade-off that can be made to increase the performance is lowering the number of intervals. The results show that the execution time increases even more rapidly when the number of intervals increases.

8. FUTURE WORK

The current C implementation does not include Wi-Fi communication. The Wi-Fi communication with the DSM system will add additional overhead, because the communication costs processing power and memory. The ESP32 might not be able to satisfy the requirement of 1 second for an interval duration of 60 seconds, resulting in 1440 intervals if the Wi-Fi implementation is added. On the other hand, it is a dual-core CPU, one core can be responsible for the communication and the other one for the schedules, this research only used one CPU core. This can be investigated in future research. Currently, basic versions of the scheduling algorithms are implemented, future work can focus on the impact of enhancements like vehicle-to-grid (V2G), where the energy from the EV batteries is sold to the grid. Price steering is another feature that can be added to the implementation. Multiple ESP32s can create a mesh network, this can be used for instance sharing the computation power or charging a fleet of EVs instead of one.

9. REFERENCES

[1] M. Babiuch, P. Foltýnek, and P. Smutný. Using the esp32 microcontroller for data processing. In *2019 20th International Carpathian Control Conference (ICCC)*, pages 1–6, 2019.

[2] Espressif Systems. *ESP32-S2 Family Datasheet*, 2021. Version 1.3, Available at `https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf`, version 1.6.0.

[3] Espressif Systems. *ESP32 Series Datasheet*, 2021. Version 3.6, Available at

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[4] G. Gaspar, P. Fabo, M. Kuba, J. Dudak, and E. Nemlaha. Micropython as a development platform for iot applications. In R. Silhavy, editor, *Intelligent Algorithms in Software Engineering*, pages 388–394, Cham, 2020. Springer International Publishing.

[5] M. Gerards, H. Toersche, G. Hoogsteen, T. van der Klauw, J. Hurink, and G. Smit. Demand side management using profile steering. In *PowerTech, 2015 IEEE Eindhoven*, pages 457759:1–457759:6. IEEE Power Energy Society, June 2015. 10.1109/PTC.2015.7232328 ; null ; Conference date: 29-06-2015 Through 02-07-2015.

[6] Y. He, B. Venkatesh, and L. Guan. Optimal scheduling for charging and discharging of electric vehicles. *IEEE Transactions on Smart Grid*, 3(3):1095–1105, 2012.

[7] G. Hoogsteen, J. L. Hurink, and G. J. M. Smit. Demkit: a decentralized energy management simulation and demonstration toolkit. In *2019 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, pages 1–5, 2019.

[8] G. Hoogsteen, A. Molderink, G. Smit, J. Hurink, B. Kootstra, and F. Schuring. Charging electric vehicles, baking pizzas, and melting a fuse in lochem. *CIRED: Open Access Proceedings Journal*, 2017(1):1629–1633, Oct. 2017.

[9] V. M. Ionescu and F. M. Enescu. Investigating the performance of micropython and c on esp32 and stm32 microcontrollers. In *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pages 234–237, 2020.

[10] M. T. Jones. Optimization in gcc. *Linux journal*, 2005(131):11, 2005.

[11] R. S. Machado, R. B. Almeida, A. D. Jardim, A. M. Pernas, A. C. Yamin, and G. G. H. Cavalheiro. Comparing performance of c compilers optimizations on different multicore architectures. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 25–30, 2017.

[12] S. Martinenas, A. B. Pedersen, M. Marinelli, P. B. Andersen, and C. Trreholt. Electric vehicle smart charging using dynamic price signal. In *2014 IEEE International Electric Vehicle Conference (IEVC)*, pages 1–6, 2014.

[13] A. D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design Test*, 34(1):77–90, 2017.

[14] M. Schoot Uiterkamp, M. Gerards, and J. Hurink. Online electric vehicle charging with discrete charging rates. *Sustainable Energy, Grids and Networks*, 25, Mar. 2021. Elsevier deal.

[15] R. G. Scottow, A. B. T. Hopkins, and K. D. McDonald-Maier. Instrumentation of real-time embedded systems for performance analysis. In *2006 IEEE Instrumentation and Measurement Technology Conference Proceedings*, pages 1307–1310, 2006.

[16] T. van der Klauw. *Decentralized Energy Management with Profile Steering: Resource Allocation Problems in Energy Management*. PhD thesis, UT, Netherlands, May 2017. CTIT Ph.D. thesis series no. 17-424.