Memory Footprint Optimization Techniques for Machine Learning Applications in Embedded Systems

Manolis Katsaragakis[⋆α], Lazaros Papadopoulos[⋆], Mario Konijnenburg[•], Francky Catthoor[α†◦], Dimitrios Soudris[⋆]
[⋆]*Microprocessors and Digital Systems Laboratory, ECE , National Technical University of Athens, Greece*
[α]*Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, 3001 Heverlee, Belgium*
[•]*IMEC-NL, Eindhoven, The Netherlands*, [†]*IMEC, Kapeldreef 75, 3001 Heverlee, Belgium.*
[⋆]mkatsaragakis, lpapadop, dsoudris@microlab.ntua.gr, [◦]francky.catthoor@esat.kuleuven.be, [•]mario.konijnenburg@imec.nl

***Abstract*—Effective memory management is an important requirement for embedded devices that operate at the edges of Internet of Things(IoT) networks. In this paper, we present a set of memory optimization techniques for machine learning applications developed in Python. The proposed techniques aim to avoid the main drawbacks of static memory allocation and to promote dynamic memory management, in order to optimize memory usage and execution latency. The results of the presented techniques are evaluated in a biomedical application, showing significant memory utilization and performance improvements (64% reduction in memory size requirements and 51% execution time reduction). Additionally, we highlight the applicability of the proposed techniques to a wide variety of IoT applications that leverage machine learning algorithms. Finally, the results of the optimized biomedical application in Python are compared with the corresponding version of the application in C and we identify trade-offs between software maintainability and memory size requirements.**

***Index Terms*—Embedded Systems, IoT, Machine Learning, Resource Management, Memory Optimization, Python, C**

I. Introduction

Embedded systems refer to a combination of computer hardware and software, extending computation beyond traditional devices, such as desktops, laptops, smart-phones and tablets, to any physical devices, everyday objects and wearables, composing the world of Internet of Things (IoT). Each platform is designed for a specific application or for specific functions within a larger system. These applications require to be executed under the corresponding embedded system's inherent constraints of strictly limited computing and memory resources, while, on the same time, the execution latency is of major importance.

Over the last years, machine learning (ML) paradigm in embedded systems and IoT is the upward trend of research, both in industrial and in academic community. Several researches have been conducted on the implementation of machine learning applications and their adjustment on embedded systems. Authors of [1] implemented an algorithm for ECG analysis and classification on IoT, using Support Vector Machines (SVM), while authors at [2] present a deep reinforcement learning framework for autonomous driving, targeting its deployment on embedded hardware.

In order to reduce the programming effort and the required in-depth understanding of the many available and emerging ML techniques, many of those applications are implemented in a high-level programming language, such as Python. Python offers a variety of well established ML libraries, such as the *Scikit-learn* [3] software library, which supports programmers in developing demanding applications, while minimizing the programming effort required. However, Python programming often imposes static memory management, leading to increased memory requirements. Thus, applications implemented using standard Python are not suitable for embedded devices and IoT, as in many cases the required memory resources skyrocket, making the execution of an application to a resource constrained device prohibitive. Therefore, porting Python applications in embedded systems is a significant challenge. Approaches that enable dynamic memory management by Python applications and optimize the overall memory utilization enable the effective deployment of such applications in embedded devices with limited memory.

There exists several systematic approaches that enhance the dynamic memory behavior of applications. Authors of [4] and [5] propose the Dynamic Data Type Refinement (DDTR) methodology, which enables the systematic customization and refinement of dynamic data types of applications. In [6], Dynamic Memory Management (DMM) is presented, which can be used to construct dynamic memory allocators, according to application requirements and system constraints. However, such approaches have been developed only for C/C++ applications. Additionally, in [7], authors present a solution for efficiently mapping arbitrary C code into hardware, while in [8] authors describe and automatic framework for dynamic data type optimization in C language. To the best of our knowledge, very limited research has been conducted so far on memory optimizations for applications developed in Python.

This paper proposes a set of memory optimizations for Python applications that leverage machine learning techniques and demonstrates the memory gains that can be obtained. The techniques focus on enabling dynamic memory usage, in order to avoid the pitfalls of static memory management, i.e. high memory footprint and increased execution latency. To demonstrate our results, we selected a real-life and biomedical application for activity classification. It is a representative application developed in Python that uses the Random Forest algorithm to provide accurate classification. **The novel contributions of this work are the following:**

- We present a set of techniques for dynamic memory

This work has received funding from the European Unions H2020 research and innovation programme under grant agreement No 801015 (EXA2PRO, www.exa2pro.eu).

optimizations for Python applications enabled by machine learning algorithms. The techniques are based on the native characteristics of the application data structures and its algorithm.
- An evaluation of the proposed techniques is presented, highlighting their ability to significantly optimize the memory management through dynamic behavior, while achieving lower execution latency compared to the static memory management approaches.
- Finally, trade-offs between maintainability of programming and memory requirements are identified.

The rest of the paper is organized as follows: Section II analyzes the memory optimization steps implemented in our techniques, the basic principles and the characteristics of the application optimized in this work. In section III we present the experimental evaluation and the results of our proposed techniques in the examined scenarios, while section IV concludes this paper and provides future challenges and directions in the dynamic memory management research area.

II. Memory optimization of machine learning Python applications

A. Proposed Memory Optimization Techniques

We propose an optimization flow consisting of repeated discrete steps, as illustrated in Figure 1.

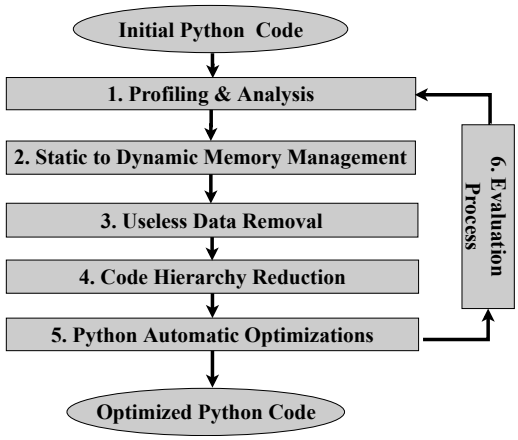

Fig. 1: Memory Optimization Steps

The input of the methodology is the original source code of the Python application. The optimization methodology consists of the following steps:

1) **Profiling and Analysis:** In order to obtain a full perspective of the application's behavior, a detailed profiling and analysis is a process of major importance. Critical spots in the code, concerning the memory consumption of a program through time, the usage of particular instructions and the frequency and duration of function calls are extracted. There already exist mature automated tools, to analyse the memory performance of Python applications, which provide information from a different point of view. More specifically, Python's *memory_profiler* [9] is used, which is a pure python module based on the *psutil* module [10], for memory monitoring of a process from operating system's perspective. Additionally, we use Python's *Guppy-PE* [11], which is a library and programming environment that offers object and heap detailed memory sizing, profiling and debugging, while the Linux tool *htop* [12] provides additional information about the running processes. All these complementary data are fed into the next steps.
2) **Static to Dynamic Memory Management:** With respect to the application's functionality and its inherent characteristics, the optimization of memory utilization through dynamic memory usage aims at avoiding the drawbacks of static memory management. Static memory management solutions are not suitable for embedded applications, as worst-case size is assumed for every data storage request, while, at the same time, efficient memory block re-usability cannot be implemented. Thus, resource requirements in terms of memory are higher than the actual required. Through the insertion of dynamic memory allocation constructs, the required memory is allocated during the run time, which enables programmers to declare only the memory required, without worrying yet about possible wastage and overheads. So this allows to keep a high level of maintainability without paying the penalty of huge overhead due to the presence of the subsequent steps.
3) **Useless Data Removal:** After a detailed profiling and analysis, there can be detected parts of code, data and imported modules that are useless in the actual functionality of the application, or that are duplicated versions of already existing parts. The former happens in cases of inefficient development by the application's developer, while the latter is, usually, caused due to the object-oriented nature of Python by creating instances of the same object in function calls. Keeping alive only the actual data and modules required, we will show that significant memory optimizations can be obtained.
4) **Code Hierarchy Reduction:** Code hierarchy refers to the different modules and sub-modules that the implemented application is divided. An important factor that affects the overall memory footprint of an application is the memory overhead added by the interpreter for the static memory allocation of modules in the beginning of each program. Python's memory manager, by default, loads statically every single module used in the application, without taking into account which functions will be actually executed at run-time. Therefore, another way to optimize the total memory footprint of an application is to minimize the overhead added by these data. In practice, without modifying Python's memory manager, moving the functionality of the actual useful code from a module inside the application and reducing code hierarchy, reduces this overhead. This has to be applied selectively though to keep the maintainability at a reasonable level while removing all relevant parts of the incurred memory footprint overhead.
5) **Python Automatic Optimizations:** Python's interpreter offers automated flags and options in order to optimize the executed application in terms of required computing and memory resources.
6) **Evaluation Process:** Last but not least, an evaluation of the applied optimizations is required. There is a possible emergence of new parts of code and data that should be

transformed. The proposed method terminates when the application is fully optimized and no other transformations can be applied, or at a specific memory/latency threshold that satisfies the programmer's and/or the embedded device's inherent requirements.

After the aforementioned steps are applied, the final output of this process is the optimized source code of the corresponding input application.

B. Representative Use Case Application

The use case application of this research is an IoT machine learning application that belongs to the biomedical wearable devices domain. In particular, the application provides activity prediction of patients, given as input sensor coordinates (x, y, z) placed on the patient's chest and a set of features per patient. The possible classification states are: lying(1), sedentary(2), dynamic(3), walking(4), running(5) and biking(6).

The application uses a pre-trained python auto-regressive model to provide predictions and the classifier used is the Random Forest(RF). RF [13] is an ensemble machine learning method that is often used for classification. Multiple decision trees are constructed, each of which makes a prediction according to the given input.

The final prediction is based on majority voting classification of the individual trees. The RF classifier is implemented in the Anaconda version of Python language using the scikit-learn software library.

III. Experimental Results and Evaluation

A. Experimental Setup

Our experiments were conducted on a machine with an Intel i7-8700 processor at 3.2GHz and a dual in-line memory module(DIMM) based on double data rate fourth generation (DDR4) RAM. We used a data-set consisted of 197,633 coordinates, which are split into data blocks of 128 coordinates each one, thus generating 1544 patients data points and 15 features per patient. The data blocks are independent (i.e. there are no data dependencies between them). Therefore, the output of the application is 1544 activity predictions, which are produced sequentially. We used 20 estimators (i.e. trees), without any limitations in terms of tree depth. All these parameters settings were derived from the real-life application context at IMEC.

B. Results and Evaluation

After a careful profiling and analysis(step 1.) and a theoretical analysis of the functionality of the algorithm, the optimizations presented in section II-A were applied in practice and they are evaluated according to their optimization impact. Figure 2 illustrates the footprint's evolution through the optimization process. The initial memory footprint was measured at 258 MB in average, all statically allocated in the beginning of the execution.

Figure 3 illustrates the comparison between the initial static memory management and the functionality of the application after applying the static to dynamic memory management transformation (step 2.). The prediction phase of the random forest algorithm is implemented as follows: As soon as the data point is ready, it is fed to all the trees of the forest and then the output of each tree is gathered to perform the

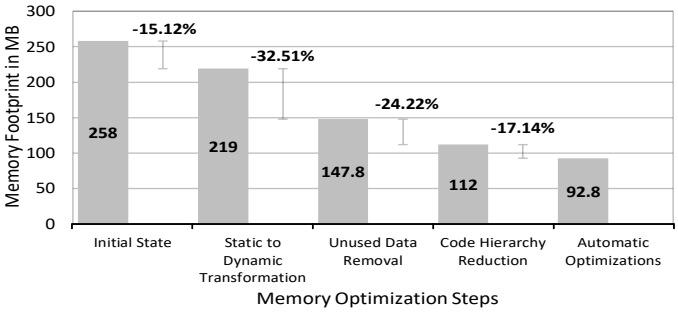

Fig. 2: Memory Footprint per Optimization Step

prediction. However, if the prediction is executed sequentially, then only the information of a single tree is required to be allocated in the memory at each point of the prediction phase. Thus, possible performance can be traded for storage requirements, respecting the functionality and the accuracy of the algorithm. Furthermore, implementing this transformation in practice showed that all classifiers information are not required. The required functionality and the necessary data can be transferred to the application and handled by the developer. Experimental evaluation showed that this optimization step reduced the overall memory footprint had an impact of 15.12% on the initial memory, which corresponds to 39 MB in absolute memory size, as shown in Figure 2.

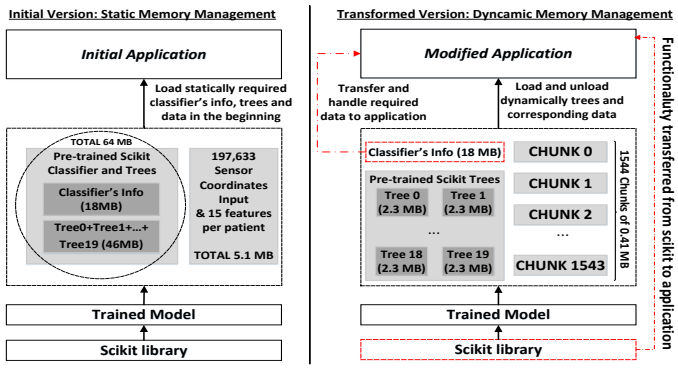

Fig. 3: High-level Static to Dynamic Transformations

Moreover, as a 2nd step (step 3.) of the memory optimization process, useless or duplicated code, data and modules that had been detected during the profiling phase, were deleted, reducing the memory footprint of the application by 71.2MB, setting it at 147.8 MB (32.51% less than the previous state, as shown in 2). As far as the code hierarchy(step 4.) reduction is concerned, in some cases the useful part functionality was transferred inside the application to avoid memory overhead by data duplication and interpreter's management, while others were not modified, providing a reduction of 35.8 MB. Finally, Python's automatic optimizations (step 5.) assisted in the optimization of memory footprint by 19.2 MB, reducing it at 92.8 MB. By summing up the results from our experiments, we extract that we achieved an overall memory improvement of 64%, so a significant factor 3 compared to the initial state of the application, as depicted in 2.

Further experiments are conducted, in order to evaluate the gains of our proposed method. An additional key finding

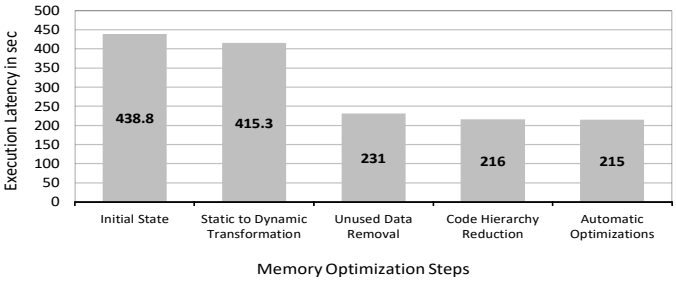

Fig. 4: Execution Latency per Optimization Step

of our research is the impact of the presented optimization method to the execution latency of the examined application. In particular, as illustrated in Figure 4, a reduction of 51% on the average execution time is observed. Memory accesses require higher execution latency, thus adding a significant overhead in the overall execution time of the application. By optimizing memory footprint, thus reducing RAM and cache accesses, the execution latency is reduced.

C. Discussion

Considering, the wide usage of real-time applications in low-resource embedded devices, we need to evaluate whether the Python language is actually suitable for devices with such limited resources. Therefore, the final version of the optimized program was implemented from scratch in C language and executed on Nvidia Tegra X1 with 4 ARM Cortex-A57 processors running at 1.9 GHz and 4 GB of RAM as an embedded device alternative.

In table I the experimental comparison between Python and C implementations is illustrated, using the same input data. C implementation outperforms the corresponding Python implementation, requiring 96.8% less memory footprint, while executing 1.5x times faster. The former is due to the nature of the Python interpreter, the implementation of the machine learning library and the available programming constructs in Python, while in C no such behavior was observed. As far as the execution latency is concerned, C is a compiler-based language. Thus, in the compilation process, several optimizations can be implemented, while there is no time overhead through the execution, in contrast to the interpreted approach of Python, in which the source code translation is done at run time. Furthermore, the C programming language allows programmer to handle efficiently the dynamically allocated data types and construct his own structs.

However, in order to achieve low memory requirements and the execution time of the C implementation, the programming effort required is significantly larger than the in the corresponding Python implementation. Therefore, trade-offs between performance and programmability can be demonstrated.

TABLE I: Python and C Code Comparison

	Memory Footprint	**Execution Latency**	**Lines of Code (LoC)**
Python Implementation	*92.8 MB*	*215 sec*	*1x*
C Implementation	*2.9 MB*	*142 sec*	*3.4x*

Python offers high level software abstractions of ready-to-use modules, which reduce the programming effort and the need for in-depth understanding of the many available and emerging machine learning techniques, in cost of increased memory requirements. On the other hand, C code requires higher programming effort and technical understanding of algorithms that are manually implemented by developers. However, the C implementation provides low memory utilization and increased performance. By using Lines of Code (LoC) as a metric of the required programming effort to develop the application, the C implementation of the biomedical application requires 3.4x more LoC that the corresponding Python version (I).

IV. Conclusion

In this paper we addressed the problem of memory management for machine learning Python applications in embedded systems. We presented novel memory optimization techniques for Python applications, achieving to avoid the main drawbacks of static memory management and promote dynamic memory transformations. The results of those techniques are evaluated, reaching an optimization of 64% in terms of memory footprint and 51% in terms of execution latency and its applicability to a variety of applications. Furthermore, the corresponding results were compared with C programming implementations, extracting trade-offs between programmability and memory requirements. Our future research goal is to examine strategically the possible expansion of our proposed approach for further optimizations and apply more automated processes.

References

[1] D. Azariadi, V. Tsoutsouras, S. Xydis, and D. Soudris, "Ecg signal analysis and arrhythmia detection on iot wearable medical devices," in *2016 5th International conference on modern circuits and systems technologies (MOCAST)*, pp. 1–4, IEEE, 2016.

[2] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] A. Bartzas, S. Mamagkakis, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris, and A. Thanailakis, "Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1, pp. 6–pp, IEEE, 2006.

[5] T. Papastergiou, L. Papadopoulos, and D. Soudris, "Platform-aware dynamic data type refinement methodology for radix tree data structures," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 78–85, IEEE, 2015.

[6] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quirós, A. Bartzas, F. Catthoor, and D. Soudris, *Dynamic memory management for embedded systems*. Springer, 2015.

[7] L. Séméria, K. Sato, and G. De Micheli, "Resolution of dynamic memory allocation and pointers for the behavioral synthesis from c," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pp. 312–319, IEEE, 2000.

[8] C. Baloukas, L. Papadopoulos, R. Pyka, D. Soudris, and P. Marwedel, "An automatic framework for dynamic data structures optimization in c," in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pp. 155–160, Sep. 2010.

[9] https://github.com/pythonprofilers/memory_profiler.

[10] https://github.com/giampaolo/psutil.html.

[11] https://github.com/zhuyifei1999/guppy3/.

[12] https://github.com/hishamhm/htop.

[13] A. Liaw, M. Wiener, *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.