A Memory Footprint Optimization Framework for Python Applications targeting Edge Devices

Manolis Katsaragakis[a,b], Lazaros Papadopoulos[a], Mario Konijnenburg[c], Francky Catthoor[b,d] and Dimitrios Soudris[a]

[a]National Technical University of Athens(NTUA), Greece
[b]Katholieke Universiteit Leuven(KUL), Belgium
[c]IMEC-NL, Eindhoven, Netherlands
[d]IMEC-BE, Leuven, Belgium

ARTICLE INFO

Keywords:
Memory Management, Edge Computing, Resource Management, Python

ABSTRACT

The advantages of processing data at the source motivate developers to offload computations at the edges of IoT networks. However, the computational resource constraints of edge computing devices limit the opportunities for deploying applications developed in high-level languages at the edge. In contrast to C/C++, applications developed in Python and other dynamic high-level languages, often require an increased amount of memory size, due to their inherent static memory management approach. Therefore, developers targeting the deployment of applications in dynamic high-level languages at the edge need support by methodologies and tools, which will allow these applications to exploit the limited memory resources efficiently. This work presents a memory optimization framework for applications developed in the Python programming language targeting edge devices. Aiming to avoid the inherent pitfalls of static memory management that Python's integrated memory manager imposes, the framework targets the reduction of the required memory footprint, by integrating a set of static and dynamic optimizations. The evaluation results, based on a set of representative real-life benchmark suites and applications, show 64% average memory footprint reduction, over the CPython's minimum baseline of 24MB. Additionally, we investigate the impact of memory size reduction on execution time and energy consumption. The results show 51% lower execution time and 47.3% reduction in the energy consumed.

1. Introduction

Over the last years, there is an ever-growing number of interconnected IoT devices, providing new services and applications in a variety of domains, such as autonomous driving, agriculture, biomedical and healthcare. As the edge-to-cloud continuum paradigm evolves, there is an increasing need of processing and storage of data at the edges of the IoT networks, which are typically based on memory-constrained embedded devices. This approach has several advantages compared to the typical offloading of computations to the cloud. Consuming data directly at the source, contributes to the energy efficiency and low latency of the whole IoT system. On the contrary, the wireless transmission of data to the cloud is typically expensive in terms of latency and energy consumption, while it may impose security issues.

Providing high Quality of Service (QoS) based on real-time computations, while operating in resource-constrained edge devices is challenging. Developers are expected to address issues such as the processing power and memory size limitations of the underlying hardware while targeting real-time performance and low energy consumption [1]. These constraints, which are often contradictory, motivated the development of several design methodologies at various abstraction levels [2].

As the edge computing paradigm evolves, the popularity of high-level programming languages increases among IoT developers. For instance, Python relies on a wide variety of libraries and abstractions which encapsulate low-level optimizations and allow the expression of more functionality with less amount of code [3, 4, 5]. These features enable rapid application development, due to the limited programming effort required. Today, Python dominates in the areas of scientific computing and machine learning in both academia and industry [6], through libraries and frameworks such as the *scipy*, *numpy* and *Pytorch*.

However, porting computationally demanding applications developed in high-level languages to resource constrained edge devices is challenging. For instance, there are several studies indicating the high energy consumption and increased memory demands of Python compared to C/C++ [7, 8, 9, 10]. Applications developed in Python

Device Name	Memory	Device Name	Memory
RT5350F-OLinuXino	32MB	Giant Board	128MB
Arduino Yún	64MB	Intel Galileo Gen 1 & 2	256MB
STM32 MCU		BeagleBone Black	
LC-CherryPi-PC-V3S			

Table 1: Memory specifications (RAM) of indicative state-of-the-art embedded boards

*This work has been partially funded by EU Horizon 2020 program under grant agreement No 101015922 AI@EDGE (https://aiatedge.eu/).

mkatsaragakis@microlab.ntua.gr (M. Katsaragakis); lpapadop@microlab.ntua.gr (L. Papadopoulos); mario.konijnenburg@imec.nl (M. Konijnenburg); francky.catthoor@esat.kuleuven.be (F. Catthoor); dsoudris@microlab.ntua.gr (D. Soudris)

need to be optimized to meet the constraints and limitations of the edge devices. In particular, Python relies on static memory management, which often leads to increased memory requirements compared to more flexible dynamic approaches, which are available in programming languages such as C and C++. Therefore, applications implemented using standard Python, often do not meet the constraints of typical embedded devices [11].

Table 1 shows the memory resources (amount of RAM) for some indicative state-of-the-art IoT devices. Applications are expected to adapt to the limited memory resources of such architectures. Even though the *microPython* implementation has been developed to target resource constrained embedded systems, the supported modules, and packages are very limited, while it is not compatible with CPython and other implementations [12]. Another feature of the CPython memory allocator, which is a result of its static approach, is that the allocated memory does not always get released back to the operating system, but instead it is offered back to the Python interpreter. The reason is that Python relies on a dedicated allocator for handling small objects that keeps memory reserved for future use. Therefore in long-running processes, we observe an incremental reservation of memory space, which may not be actually utilized, leading to increased memory fragmentation.

In order to deploy complex Python applications, such as machine learning algorithms and neural networks to the edges of IoT networks developers need methodologies and tools that will reduce the memory demands of these applications. Towards this direction, this work presents a novel framework that integrates a set of memory footprint optimization techniques for Python applications targeting IoT-embedded devices. The framework combines both static and dynamic optimizations applied at the application-level, which significantly reduce the memory requirements of Python applications. The novel contributions of this work are the following:

- **The development of a systematic methodology for memory footprint optimizations for Python applications targeting resource-constrained embedded devices.** The methodology is based on a set of optimization techniques, which address the inherent pitfalls of static memory management on which the Python's default memory management relies. The methodology is supported by two novel tools that automate the process of redundant data removal and reduction of code hierarchy, which have a significant impact on the Python applications' memory requirements.
- **A thorough evaluation of the proposed memory optimization techniques,** demonstrated in a wide variety of representative real-world Python applications. Apart from demonstrating their effectiveness in terms of memory utilization, we also investigate their impact on their real-time performance.

The rest of the paper is organized as follows: Section 2 summarizes the related work on memory management optimizations for embedded systems and Python applications. An overview of the Python memory management approach and the motivation for the existing work is presented in Section 3. Section 4 describes the proposed methodology in detail. The evaluation results and a discussion highlighting the main observations are presented in Section 5. Finally, in Section 6 we conclude this work.

2. Related Work

Several works in the existing literature propose memory management optimizations for embedded systems. Authors of [13] and [14] propose the Dynamic Data Type Refinement (DDTR) methodology, which enables the systematic customization and refinement of dynamic data structures for embedded applications, in terms of performance, memory footprint, and energy consumption [11]. A complementary methodology is the Dynamic Memory Management (DMM), which proposes customized dynamic memory allocators, to meet the application requirements and embedded system constraints, in terms of performance and memory footprint [15]. However, these approaches have been developed for C/C++ embedded applications, while Python and other modern high-level languages are not supported.

A few recent works propose memory management optimization methodologies targeting the elimination of memory leaks and fragmentation. For example, in [16] the authors leverage machine learning techniques for the automatic detection of memory leaks in C/C++ codes, while in [17] is proposed a replacement for malloc/free aiming to eliminate the memory fragmentation of C/C++ applications. A similar approach utilizes machine learning techniques to reduce the memory fragmentation in C++ server workloads, induced by long-lived objects allocated at peak heap size [18]. The aforementioned approaches target C/C++ applications, and they cannot be directly mapped to Python applications, due to the different structure of the memory allocation in dynamic languages, compared to the static.

With regard to Python, several works focus on analyzing Python's behavior and memory management performance. The authors of [19] provide a detailed quantitative analysis of the overhead in Python without and with just-in-time (JIT) compilation and they identify the existing memory overheads due to Python's interpreter and built-in memory manager, while also discussing the limitations of Python's garbage collector. Similarly, the authors of [20] study the Python's interpreter and calculate the overheads of some of the language's features such as the dynamic typing and reference counting. These works focus on identifying the overhead of Python features in terms of performance and memory and they do not propose optimization techniques. Therefore, they can be considered as a first step towards a more through understanding of Python's inherent limitations in terms of memory management.

Regarding the memory management optimizations for Python, the authors of [21] propose a Python package for

Research Goal	Related Works
C/C++ Memory Management	[13, 14, 15] [29, 16, 17, 18]
Qualitative/Quantitative Analysis	[19, 20, 30]
Interpreter-oriented Optimizations	[23, 24]
Novel Garbage Collectors	[25, 26, 27]
Library and Application-specific Optimizations	[31, 21, 22]
Application-Level Memory Footprint Optimizations	Present work

Table 2: Summary of related research for Python memory footprint optimizations

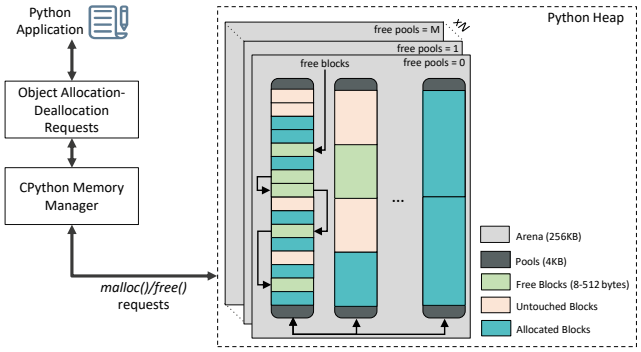


Figure 1: CPython's memory management organization

reducing the performance and memory overhead of loading, traversing, and manipulating tree data structures in Python applications. The authors of [22], propose a Python-based optimization framework for high-performance genomics, that combines the advantages of high-level languages such as Python with the performance of lower-level languages such as C/C++. In order to overcome the inherent limitations of the Python interpreter and memory management approach, the work in [23] proposes an alternative bytecode interpreter, based on CPython. Aiming to enable the efficient deployment of Python applications in embedded systems and micro-controllers, the authors of [24] introduce a run-time system that allows the execution of high-level languages and they discuss the static memory allocation behavior of Python. Additionally, the work in [25] proposes an abstract garbage collector (AGC), which detects and removes unreachable abstract addresses and works as a tracing garbage collector. A similar approach in [26] describes a dynamic memory leak detection method for dynamic languages. The work in [27] introduces a garbage collector, which is enabled by machine learning methods to allow efficient memory deallocation. The authors of [28] propose a hardware-software co-optimization memory management scheme for dynamic languages, including Python.

The aforementioned approaches that target Python can be considered complementary to the optimizations proposed in this work. Although the existing research has highlighted the effectiveness of memory footprint optimization for Python libraries, no work to date, to the best of our knowledge, has proposed a memory footprint optimization framework that specifically targets Python applications deployed on edge computing devices. The proposed framework is complementary to several existing works and can further be extended, for example by combining advanced Python garbage collectors. Table 2 summarizes the related work and highlights the position of the present work in relation to the existing literature.

A preliminary version of the proposed methodology has been presented in [11]. However, the present work includes significant improvements and extensions, including tool support, full automation and an extensive description of several steps, such as the "Redundant Data Removal" and "Code Hierarchy Reduction", as well as thorough evaluation and explanation of results based on new benchmarks.

3. Background and motivation

Across the different Python implementations, various approaches in terms of the way that memory management is performed have been introduced, however, the core principles remain the same. In this work, we select the CPython version of Python, as it is the default implementation provided by Python developers [32], thus the most widely utilized version of the various implementations. We focus on the special internal characteristics and limitations of CPython's memory management.

Typically, in CPython, the code gets compiled to a low-level, platform-independent intermediate representation of the source code, namely the *bytecode*, which is finally loaded and executed by the Python interpreter engine. The Python memory manager relies on a garbage collector, which employs a reference counting technique for monitoring the active memory objects. With reference counting, the runtime keeps track of all of the references to an object. When an object has zero assigned references, it cannot be accessed by the program code anymore, so its allocated memory can be freed and returned to the system. Even though the automatic garbage collection is faster and hides implementation details, it imposes computation and memory overhead in order to track all application references.

Furthermore, memory leaks occur in practice: the reference count of non-active objects may never become zero, even though these are not used in any subsequent computations. Typical causes of memory leaks are uncommon paths through the code structure. For instance, a function may allocate a block of memory and then free the block again, while the application algorithm does not use the specific block. Moreover, early termination of the application due to an error condition can lead to a memory leak. In general, if a Python application terminates early, all of its memory is released back to the operating system, however, there exist scenarios that lead to memory leaks. For instance, Python applications that utilize finalizers [33], which are special methods that are called when an object is about

to be destroyed, may not be called if the program terminates abnormally. Moreover, the integration of third-party libraries into Python applications may not efficiently release the memory allocated, when the Python program terminates early.

Python's heap management is implicit to the application's developer and handled by the interpreter, i.e. the developer has no direct control over it. The dynamic memory allocation/deallocation on Python's heap space is performed by the integrated memory manager through the Python/C API functions [34]. The Python memory manager is organized into three layers [35]:

- **Blocks:** Blocks refer to the fundamental memory management component, which is defined by a continuous number of bytes of virtual memory. Each block can be characterized as *untouched*, i.e. a portion of memory has not been allocated yet, *free*, i.e. an allocated block that has been "freed" and has no relevant data and *allocated*, which contains data required by the Python application. The blocks that are deallocated by the memory manager (i.e. they are "freed"), are added to a list data structure for future use by the manager (freelist). Typically, the size of the blocks varies from 8 to 512 bytes and it is a multiple of eight.

- **Pools:** Pools consist of blocks from a single-size class. Whenever a block is requested, the memory manager checks the pools for that block size. Each pool can be characterized as *free*, *used* or *empty*. Normally, the size of the pool is equal to the size of a memory page. Limiting the pool's size to the fixed size of blocks has a positive impact on memory fragmentation.

- **Arenas:** Finally, arenas are the highest level of memory components and contain pools of memory. Arenas are sorted by the CPython memory manager in ascending order, based on the availability of free pools in each arena, and, unlike pools and blocks, they do not have any explicit state. The typical size of the arenas is fixed at 256KB, which corresponds to the size of 64 pools, considering a 4KB page size.

Fig. 1 illustrates the organization of blocks, pools and arenas of CPython's memory manager [36]. The memory manager handles requests, by trying to provide memory from the arena with the lowest amount of available memory (i.e. the one which is mostly occupied). An interesting feature of CPython is the fact that the memory manager returns to the OS not pools or blocks, but arenas, only. In contrast to C/C++, when a memory block is deallocated, that memory is not actually returned back to the OS, but remains in a freelist. Although this approach typically reduces the memory allocation overhead in terms of performance, it is very static, leading to higher memory footprint utilization throughout the execution of an application. For IoT applications deployed on edge devices with memory constraints, such a static approach is often a significant limitation.

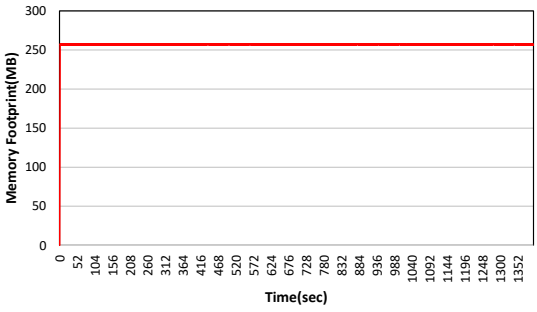


Figure 2: Motivational example showing CPython's static memory management approach

Figure 2 shows a typical memory utilization plot of a CPython application (a machine learning application derived from the IoT wearables domain [11]). Although the application is very dynamic, in the sense that it constantly allocates and deallocates memory blocks of various sizes, these fluctuations are not reflected in the OS. Even though mechanisms such as GC and Python's memory allocator are running in the background, this is not frequently observed from the operating system's perspective, thus leading to static memory utilization. For instance, the memory of objects freed by the GC are never becoming visible to the OS, as for a portion of memory to be given back to the OS, a whole arena should be freed. Therefore, from the OS's and system's perspective, the application is static it terms of memory requirements. Initially, we observe a spike for modules and data allocation, which is reserved for the rest of the execution, although many of the allocated parts are either tracked or removed by the garbage collector.

To address the aforementioned limitations, which are critical for CPython applications targeting edge computing devices, we propose a flow of application-level memory footprint optimizations integrated into a framework, which will enable a more dynamic memory management approach, to enable the efficient utilization of the limited memory resources of edge devices.

4. Memory Footprint Optimization Methodology

4.1. Overview

This section presents the systematic methodology for memory footprint optimizations for embedded Python applications and the tool-flow that supports the methodology. The input of the methodology is the source code of the application under optimization. An overview of the optimization techniques is presented in Fig. 3. The methodology consists of the following steps:

1. **Application-Level Analysis:** The first step of the proposed methodology includes the dynamic analysis of the application using well-established Python profiling tools, such as memory profiler [37], guppy [38], which enable the identification of the most critical

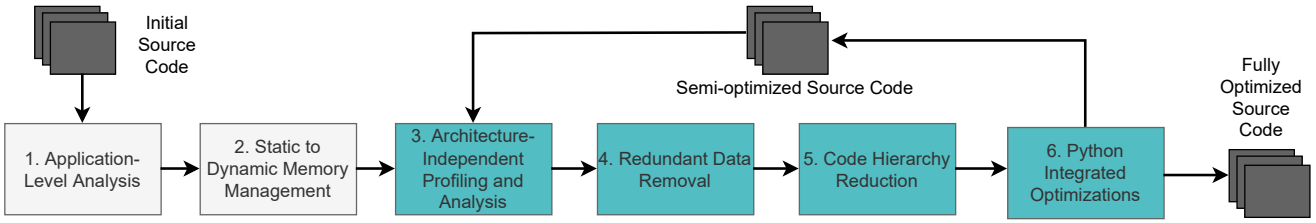


Figure 3: Overview of the memory footprint optimization methodology

application data structures in terms of memory requirements. This step identifies the source code functions and data structures, in which the applied refactoring and optimizations of the subsequent steps of the methodology are expected to have an increased impact.

2. **Static to Dynamic Memory Management:** In this step, the application source code is refactored, by converting the most critical data structures in terms of memory requirements, as identified in the previous step, from statically declared into dynamic. Apparently, this is a very application-specific step. Automating the refactoring is very challenging, because algorithmic adaptations may be needed in several cases, to avoid altering the functionality of the application. Therefore, its efficient implementation is left to developers, for now. However, the conversion from static to dynamic memory management is critical, since (i) it addresses the static memory management limitations in terms of memory utilization, as explained in the motivational example of Section 3 and (ii) it enables further memory optimizations. Expressing a significant amount of dynamic application behavior in terms of memory utilization normally enables several optimization opportunities, which are described in the next steps of the methodology.

3. **Architecture-Independent Profiling and Analysis:** After applying the necessary source code refactoring which creates a version of the application with a dynamic memory behavior, the application is profiled again. Object-trace profiling tools, such as the *Guppy-PE* [38], enable the collection of object and heap detailed memory sizing information. The profiling data generated in this step are fed as input into the subsequent memory optimization steps. Additionally, this step reveals the effectiveness of the refactoring applied during the *Static to Dynamic Memory Management* step.

4. **Redundant Data Removal:** The profiling analysis conducted by the previous step is used to identify parts of code, data, and imported modules that are useless for the actual functionality of the application, or that are duplicated versions of already existing parts. The former happens in cases of inefficient application development, while the latter is usually a result of the object-oriented nature of Python, in which the interpreter often creates instances of the same object in function calls. Keeping alive only the actual data and modules which are necessary for the application execution, enables significant gains in terms of memory footprint. The Redundant Data Removal step integrates techniques that rely both on static and dynamic analysis of the application under optimization. The implementation of this step in practice often requires extended source code refactoring. In this work, we fully automated this step. The corresponding tool which implements this step is described in detail in Section 4.2.

5. **Code Hierarchy Reduction:** An important factor that affects the overall memory footprint of Python applications is the memory overhead added by the interpreter for the static memory allocation of modules at 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 source code maintainability at a reasonable level while removing all relevant parts of the incurred memory footprint overhead. Again, this involves non-trivial reorganization of the code. Therefore, we fully automated this step and the corresponding tools we developed and integrated in the proposed framework are described in detail in Section 4.2 and Section 4.3.

6. **Python Integrated Optimizations:** Python's interpreter offers several flags and options to optimize the application execution in terms of required computing and memory resources. More specifically, we take advantage of *-O* flags to remove assert statements and discard docstrings. By implementing such optimizations in practice, an optimized Python bytecode is generated, i.e. a platform-independent low-level implementation of the target application to be executed

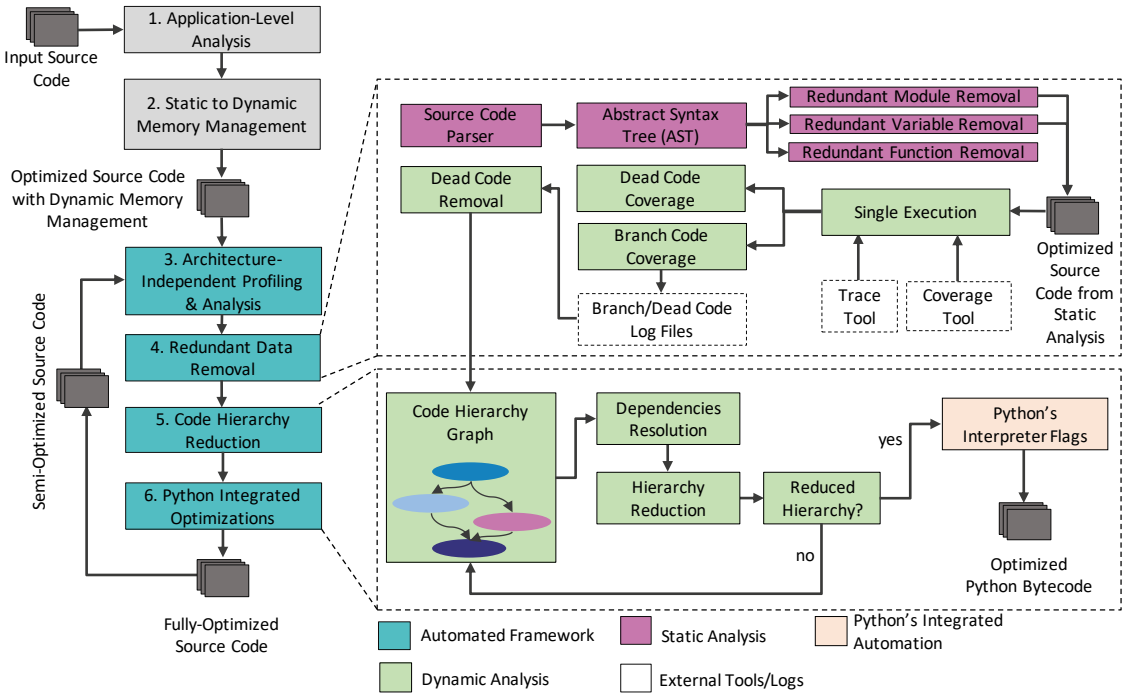


Figure 4: Memory footprint optimization methodology

on the Python's virtual environment. This step is fully automated, as it is provided directly by the Python interpreter.

The output of the methodology is the optimized Python application source code in terms of memory footprint. Steps 2, 4 and 5 are expected to have the most significant impact on the memory footprint reduction. As stated earlier, step 2 is applied manually, while steps 4 and 5 are supported by novel tools which are integrated into the proposed framework. A detailed view of the methodology is shown in Fig. 4.

4.2. Memory Footprint Optimizations based on Static Analysis

Static memory footprint optimizations refer to a set of optimizations that are statically executed. Static analysis optimizations are integrated in Step 4 of the flow, as shown in Fig. 3. The main goal of the applied static optimizations is to detect and remove redundant imported modules and variables that do not contribute to any functionality throughout the execution of the Python application. Moreover, these optimizations act as a pre-processing step for the subsequent dynamic optimizations.

To implement the algorithm, we leverage a set of PyPi's projects, namely *pyflakes* [39] and *autoflake* [40]. *Pyflakes* analyzes programs statically, detects errors and operates by parsing source files. *Autoflake* utilizes *pyflakes* and it removes unused imports and variables from Python code. By default, *autoflake* removes only redundant imports for modules that are part of the standard library. This is due to the fact that other modules may have side effects that make them unsafe to be removed automatically. In our approach, we significantly extend the functionality of *pyflakes* and *autoflake*, so that we can detect redundant or duplicated imports and data beyond the standard library, as well as detect unused functions.

In practice, this is achieved by creating the Abstract Syntax Tree (AST) of the corresponding application. AST generates the abstract syntactic structure of the source code, where every single node denotes a construct. The source code is parsed once and the AST is generated. Next, all nodes of the generated AST are traversed. We distinguish three alternative types of AST nodes, namely (i) *Import Nodes*, which refer to AST nodes related to importing modules, (ii) *Variable Nodes* which correspond to nodes that refer to object initialization and, finally, (iii) *Function Nodes*, which are related to function definitions throughout the source code. For the former, we detect the number of nodes that depend on our target *import node*. If there exist no referenced nodes, the *import node* is redundant, i.e. it is not utilized by the rest of the target application. Therefore, the examined node and the corresponding source code can be safely removed. With regard to the *variable nodes*, each variable is represented as a Python object in AST. Variable references are utilized either to load the current, assign a new value, or delete the object. Initializing a new variable is characterized as a single load or an assign, depending on the kind of initialization. If no other references exist to the target object, it is characterized as redundant and, thus, it can safely get removed from the source code.

The detection and removal of redundant *function nodes* is not straightforward. Existing tools for static analysis do not support any functionality for deleting unused functions. Therefore, we extended the functionality of the existing

Algorithm 1: Memory Footprint Optimizations Algorithm based on Static Analysis

```
   Data: Python Source Code (pythonApplicationFiles)
1  Static-Analysis(pythonApplicationFiles):
2      parsedCode = parseSourceCode(pythonApplicationFiles)
3      AST = createAST(parsedCode) /* Create Abstract Syntax
         Tree(AST) */
4      /* Traverse AST */
5      for node in AST do
6          if node is import then
7              if node has no kid nodes then
8                  /* Unused import detected */
9                  remove(node,pythonApplicationFiles,
10                 optimizedCode)
11         else if node is variable object then
12             /* If variable is not referenced for load or assign */
13             if node.loadReferences == 0 &&
                 node.assignReferences == 0 then
14                 remove(node,
                     pythonApplicationFiles,optimizedCode)
15         else if node is function definition then
16             /* Find AST nodes that reference to the current
                 node */
17             node.num_of_references = 0
18             for newNode in AST do
19                 if newNode.reference == node && newNode
                     !=node then
20                     node.num_of_references += 1
21             /* If there is no reference to this function from
                 other objects, remove function */
22             if node.num_of_references == 0 then
23                 remove(node,pythonApplicationFiles,
24                 optimizedCode)
25     return optimizedCode
```

Algorithm 2: Memory Footprint Optimizations Algorithm based on Dynamic Analysis

```
   Data: Python Source Code Files(pythonApplicationFiles)
1  Dynamic-Analysis(pythonApplicationFiles):
2      /* Single Execution */
3      while executeApplication() do
4          /* Trace application throughout the execution */
5          traceFile = traceApplication()
6          /* Generate code hierarchy tree */
7          hierarchyLayers =
             generateHierarchyTree(pythonApplicationFiles)
8      /* Code Coverage */
9      coverage = monitorCodeCoverage(traceFile,
         pythonApplicationFiles)
10     for code in coverage do
11         if code is in non-executed branch then
12             /* Avoid Branch Code Coverage */
13             continue
14         else
15             /* Remove Dead Code */
16             deleteCode(pythonApplicationFiles,
                 optimizedCode)
17     /* Hierarchy Reduction */
18     for layer in hierarchyLayers do
19         for node in layer do
20             /* First, check for intra-layer dependencies */
21             if layer has intra-dependencies then
22                 for dependentNode in layer do
23                     if dependentNode depends on node then
24                         /* Resolve intra-layer dependencies
                             */
25                         transferCode(node, dependentNode,
                             pythonApplicationFiles,
                             optimizedCode)
26             for dependentNode in layer-1 do
27                 /* Check for inter-layer dependencies with
                     layer-1 */
28                 if dependentNode depends on node then
29                     /* Resolve inter-layer dependencies */
30                     transferCode(node, dependentNode,
                         pythonApplicationFiles,
                         optimizedCode)
31     return optimizedCode
```

tools for *function node* removal. Initially, we detect all the *function nodes*, by traversing the AST. Next, we traverse again the AST and monitor all the references to object attributes or symbol names in the whole AST, which can be the function's potential callers. The *function nodes* that are not referenced by any other node in the source code, are marked as unused and, therefore, get removed from the source code. Self-references are excluded during this monitoring process, in order to be able to detect and remove recursive functions. Finally, Python offers dynamic code modifications and dynamic function definitions. Since the characterization of these cases as necessary or redundant is even more challenging, they are not considered during the static analysis process and their characterization is left as a future work.

To minimize the overhead of static analysis we provide in-memory node logging and tracing, thus avoiding extra disk storage for log files and time-consuming post-processing. Moreover, in addition to memory footprint reduction, removing redundant modules, data and functions leads to improved source code quality, as it increases code maintainability. Algorithm 1 demonstrates the core functionality of memory footprint optimizations based on static analysis.

4.3. Memory Footprint Optimizations based on Dynamic Analysis

This section describes in detail the optimizations enabled by the dynamic analysis, i.e. by profiling and executing the corresponding Python application. The optimized source code generated after applying the static analysis optimizations to remove redundant modules, variables, and functions (Section 4.2) is forwarded to the dynamic analysis for further optimizations. In particular, the dynamic optimizations include Redundant Data Removal enabled by dynamic analysis (Step 4 in Fig. 3) and the reduction of the code hierarchy (Step 5 in Fig. 3). The pseudocode of Algorithm 2 illustrates the core functionality of both optimizations, which are analyzed in the rest of this section.

4.3.1. Redundant Code Removal

In contrast to static programming languages, such as C/C++ where memory allocation and deallocation are directly done at run-time on the corresponding code line, the Python's memory manager does not dynamically (de)allocate the required memory. Explicit memory management usually leads to higher memory requirements, as a significant amount of the estimated footprint is pre-allocated statically by the memory manager. Thus, the existence of redundant code and data can cause high static memory pre-allocation, which cannot be directly deallocated by developers.

To further identify parts of the source code which can be safely considered redundant, we rely on dynamic analysis results. Throughout the execution of the application under optimization, we monitor the program execution on a different thread and trace the sequence of the source code instructions executed. We generate annotated traversed source code listings, caller/callee relationships, and list the functions executed. Moreover, by taking advantage of *coverage.py* project [41], which is a third-party coverage tool, we extract the code coverage, i.e. the percentage of source code actually executed for a particular run and the corresponding source code lines. Existing tools, such as [41, 42] do not provide code removal features. Instead, they only report the coverage, as well as a percentage indicating the corresponding reliability. Therefore, we extend their functionality, to characterize alternative parts of code and provide automated code removal features. Through our implementation, we derive a detailed code coverage report and distinguish the following two cases:

- **Dead code coverage:** These parts of source code refer to all the non-executed code apart from the ones mapped to branch code coverage. As no condition can allow the execution of these parts of source code, it is automatically removed by deleting the corresponding source code lines. Interestingly, Python allows for classes to be dynamically modified at runtime, and evaluate arbitrary Python expressions from a string-based or compiled-based-code input by using the *eval()* function. Although Python's *eval()* is a very useful function, the code generated at runtime cannot be known a priori, thus these parts of code should not be removed. However, if the application input is fully deterministic, then it may be possible to characterize the corresponding parts of the source code as redundant, too.

- **Branch code coverage:** It refers to the non-executed parts of code placed under branches. As a branch we consider every line of code that triggers switching execution to a different instruction sequence as a result of executing a branch instruction. Branch code coverage depends on control flow structure and triggering inputs, with more deeply nested statements being significantly less likely to be covered [43]. The most representative cases are *if* branches, *for*, and *while* loops. Such parts of the code are not considered as redundant, thus they are not automatically removed. Similarly to the dynamically modified dead code detected in *Dead code coverage*, such parts of code can be removed for fully deterministic application inputs, only.

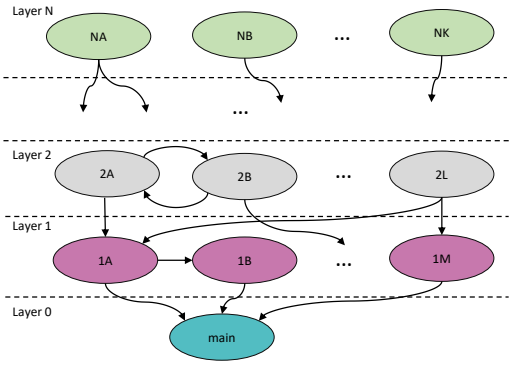


Figure 5: Code hierarchy, inter and intra-dependencies

After the identification and removal of redundant code through dynamic analysis, the optimized source code is provided as input to the *Code Hierarchy Reduction* step.

4.3.2. Code Hierarchy Reduction

An assumption on which the specific optimization relies on, is that the source code of the imported modules exists locally; either in the application's file directories or on Python's standard libraries on the device, and it is not stored as bytecode. In order to apply this optimization in practice, we generate the code hierarchy graph of all the modules for all the application layers, during the dynamic analysis. Fig. 5 illustrates a generic graph representation of a multi-layer code hierarchy. Each node corresponds to a unique module, while each edge represents a dependency among modules. For example, in Fig. 5 module $1A$ depends on module $2A$. Additionally, the same node can exist in alternative layers of the hierarchy, while there is no limitation on the number of dependencies that each node can have, i.e. multiple nodes can depend on multiple nodes. Finally, directly or indirectly, all imported modules refer to the main module of the application under optimization. Moreover, Python allows circular dependencies between modules, where each module is defined in terms of the other (i.e. module A requires module B and simultaneously module B requires module A). In fact, in the majority of cases, circular dependencies occur due to poor application development and they are often a cause of memory leaks, infinite *import* recursions and may have negative impact on the source code maintainability. To address these issues, we developed a tool which merges the corresponding modules in one, while respecting the sequence of inter-module function calls and data types access.

In order to retain the application functionality, the tool respects the dependencies between the modules. We implemented a top-down hierarchy reduction design, starting from Layer N, where there exist no dependencies from upper

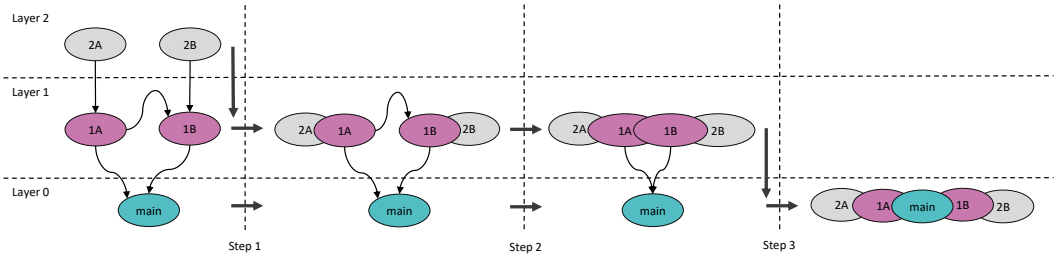


Figure 6: Example of 3-layer code hierarchy reduction

layers. We identified two distinct cases for module dependencies, namely intra-layer and inter-layer dependencies. The former refers to dependencies within the same layer, while the latter refers to dependencies among alternative module layers. Initially, we resolve the intra-layer dependencies of *Layer* N, i.e. the corresponding source codes of the independent layers are transferred to the dependent layers. Next, we proceed with the inter-layer dependencies between *Layers* N and $N-1$, in which, similarly, the source codes of *Layer* N is transferred to the corresponding dependent nodes of *Layer* $N-1$. The *Layer* $N-1$ becomes the top-level layer of the code hierarchy. The aforementioned process is repeated, until the procedure reaches *Layer* 0, where the *main* module is located, which is responsible for the application's execution.

Fig. 6 illustrates a simple example of *3-Layer* hierarchy application. Starting from *Layer* 2, as there exist no intra-layer dependencies, the functionality of nodes $2A$ and $2B$, is transferred to modules $1A$ and $1B$, respectively (Step 1). Next, the process is continued on *Layer* 1, where there exists an intra-layer dependency among the combined nodes $1A2A$ and $1B2B$. Thus the corresponding source code is transferred to the dependent node, i.e. from $1A2A$ to $1B2B$ (Step 2). Finally, as no intra-layer dependencies exist, the new node is relocated to the *main* node in *Layer* 0 (Step 3.) and the code hierarchy is fully reduced, thus the interpretation overhead due to module allocation is reduced to minimum. The core functionality of the code hierarchy reduction tool is summarized in lines 17-30 of Algorithm 2.

Even though code hierarchy reduction is expected to reduce the memory footprint, trade-offs can be identified between the source code maintainability and memory footprint. The process of hierarchy reduction is executed recursively over the alternative modules and libraries of the target application and is finalized when the memory footprint is not reduced more than a user-defined percentage $\lambda\%$, which is configurable by the user.

5. Experimental Results and Evaluation

5.1. Experimental Setup

The experiments were conducted using the Anaconda version of Python 3.7 language based on CPython implementation. We utilize a Raspberry Pi4 Model B, equipped with Quad core Cortex-A72 (ARM v8)@1.8GHz and 4GB of DRAM as a state-of-the-art embedded board. Raspberry Pi4 is one of the most widely used embedded boards over a wide variety of application domains [44]. Moreover, the memory management of Python applications, is directly handled by the Python's engine, thus the memory gains are expected to remain the same and are representative for a wide range of devices, including IoT devices, with even lower memory capacity.

Concerning the profiling and analysis (step 3 in Fig. 4), we rely on well-established tools, such as Python's *memory_profiler* [37], which is a pure Python module for monitoring the memory consumption of processes, as well as line-by-line analysis of memory consumption for Python programs. For the memory monitoring of a process from the operating system's perspective, we use the *Linux perf* [45], a profiling tool that provides an abstraction of the underlying hardware and provides low-level system metrics.

The proposed memory footprint optimization framework is extensively evaluated based on the following applications and benchmarks:

- A set of benchmarks from the *pyperformance* [46] suite. The specific benchmark suite collects real-world Python applications relevant to all Python implementations. The input of each selected *pyperformance* benchmark is the default (i.e. the one that the suite provides).
- A real-world Convolutional Neural Network(CNN) application, consisting of multiple convolution and activation layers for digit recognition, derived by [47]. As input, we provide the MNIST dataset [48], which consists of images of digits ranging from 0 to 9, represented as a matrix of shape 28x28x1.
- A real-world machine learning-enabled *IoT biomedical application* which is executed on a wearable device. 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 [49] is an ensemble machine learning method that is often used for classification. Multiple decision trees are constructed, each of which predicts

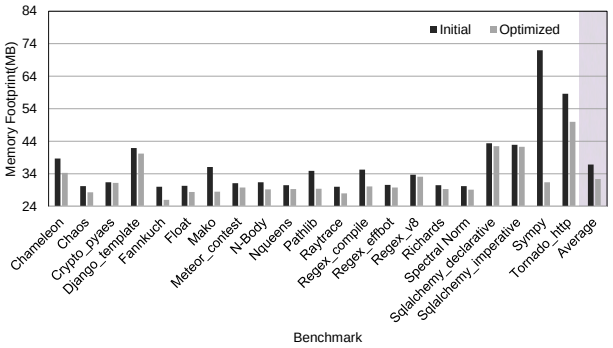


Figure 7: Memory footprint optimization for Pyperformance Benchmark Suite

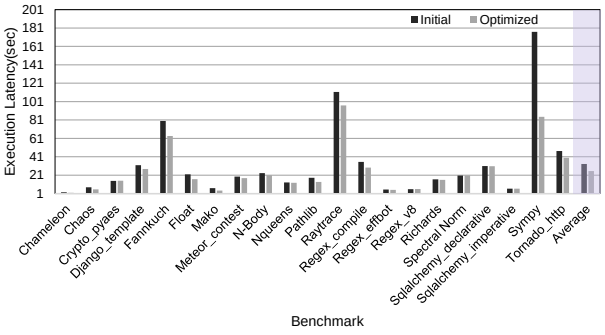
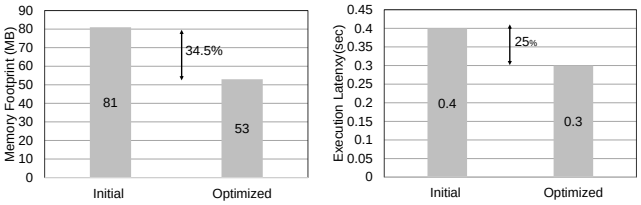


Figure 8: Execution time optimization for Pyperformance Benchmark Suite



(a) Memory footprint (b) Execution Latency

Figure 9: Experimental evaluation of memory footprint and execution latency optimizations on Convolutional Neural Network(NN) application over MNIST dataset.

according to the given input. The final prediction is based on majority voting classification of the individual trees. The RF classifier is implemented using the scikit-learn software library. We used a data-set consisting 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 parameter settings were derived from the real-life application provided by IMEC [50].

Experiments showed that the minimum possible memory footprint required for an empty Python application is 24MB. This size corresponds to a set of 96 arenas of 256KB, which are allocated at the initialization of every application, in order for the interpreter to operate efficiently, and are never deallocated until the end of the execution. Therefore, this threshold is used as a baseline for our experiments.

5.2. Pyperformance suite & CNN Application

Memory footprint optimization evaluation: Figure 7 illustrates the memory footprint optimization results before and after applying the proposed framework on each application of the *pyperformance* benchmark suite. X axis denotes each benchmark, while Y axis denotes the maximum memory footprint of the application in MB. The results show that the average memory footprint reduction is 34.6%. The benchmark with the highest memory footprint reduction is *Sympy*. *SymPy* [51] is a Python library for symbolic mathematics. Detailed analysis of the specific benchmark shows that the *Sympy* library allocates a large amount of redundant data, while in practice it utilizes only a small amount of them. Furthermore, *Sympy* is known to behave insufficiently when the user requests the handling of very long expressions [51]. The reduction of code hierarchy and the removal of useless data/code resolves the aforementioned issues, reaching a memory footprint optimization of 84.5%. Furthermore, from Figure 7 we observe that the benchmarks *Django_template*, *Tornado_http*, *Sqlalchemy_declarative* and *Sqlalchemy_imperative* show the smallest optimization in terms of memory footprint. In order to operate efficiently, all four of these require a large set of input data that need to be processed at runtime. Although such a dataset was not available in the *pyperformance* suite during the evaluation of the framework, we can safely assume that a larger input dataset will lead to increase memory footprint reduction.

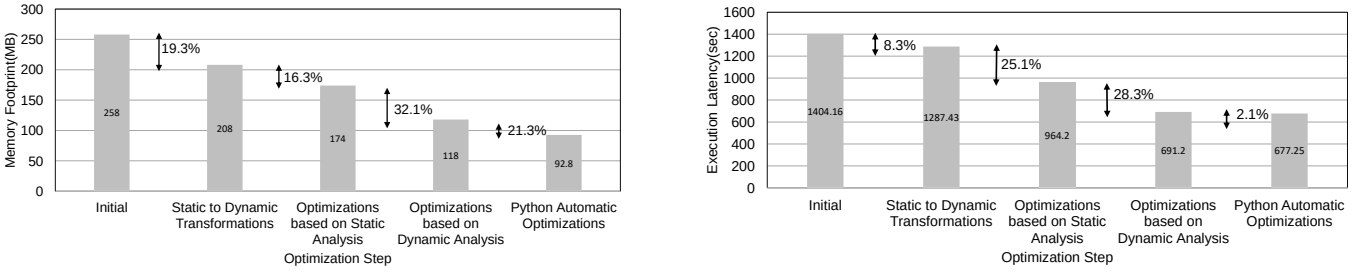
Impact on execution time: Figure 8 shows the execution time of each *pyperformance* benchmark before and after applying the memory optimization framework. The Y axis represents the execution time of the corresponding benchmark in seconds. Interestingly, the memory footprint optimizations result in execution time reduction, by 33.1% on average. We assume that the main reason is that removing redundant modules and reducing the code hierarchy simplifies the application data flow, having a significant positive impact on the execution time.

CNN application evaluation: Moreover, our experiments are also conducted on the CNN application over the MNIST dataset for embedded systems. Figure 9 illustrates the memory footprint and execution time optimizations, showing an improvement of 27MB, which corresponds to 49.1% decrease, having 24MB as baseline, as illustrated in Figure 9a. Regarding the execution latency experiments, Figure 9b presents the execution latency of the inference throughout the execution and we observe a speedup of 1.3x.

Heap memory management utilization analysis: For a more thorough analysis, we monitored the impact of the

Benchmark	Initial Average Num. of Arenas	Optimized Average Num. of Arenas	Initial Average Num. of Pools	Optimized Average Num. of Pools
Chameleon	155	137	623	512
Chaos	120	113	492	441
Crypto_pyaes	127	125	516	489
Django_template	168	161	673	636
Fannkuch	120	104	487	403
Float	121	113	488	421
Mako	144	114	587	459
Meteor_contest	124	119	503	477
N-body	126	116	511	467
Nqueens	122	117	495	469
Pathlib	140	118	563	470
Raytrace	120	112	485	448
Regex_compile	141	120	571	469
Regex_effbot	122	119	493	477
Regex_v8	135	132	544	523
Richards	122	117	491	469
Spectral Norm	121	116	485	449
Sqlalchemy_declarative	174	170	694	669
Sqlalchemy_imperative	172	169	686	677
Sympy	288	125	1139	465
Tornado_http	234	200	936	772
Average	**147.3**	**129.5 (-9.6%)**	**589.1**	**518.2 (-12.07%)**

Table 3: Python arenas and pools optimization for Pyperformance Benchmark Suite



(a) Memory footprint optimization

(b) Execution time optimization

Figure 10: Experimental evaluation of memory footprint optimizations on IoT biomedical application

optimizations on CPython's internal heap memory management. Table 3 illustrates the average number of allocated arenas and pools throughout the execution of each benchmark before and after the implementation of the framework. The experiments were conducted with a 256KB arena size. The pool size is set to 4KB, in order to be equal to the operating system's page size, aiming to reduce fragmentation; thus saving memory space. By applying the proposed framework, the average number of active arenas is reduced by 9.6% on average. Similarly, the average number of active memory pools is reduced by 12.07% on average. The application-level memory footprint optimizations lead to optimized results in terms of pool and arenas utilization. Also, by exploiting the dynamic memory aspects of an application, more pools are reduced throughout the execution, in contrast to static memory management, where pools cannot be frequently deallocated. Moreover, though the *Useless Data Removal* step, small-sized objects and variables are removed, thus leading to fewer allocations of small-sized memory blocks.

5.3. IoT biomedical application and use-case demonstration

The memory optimization methodology is evaluated on a real-world biomedical application, which is based on an RF classifier. We present a use-case demonstration of each framework's step, aiming to provide a simple example of how our approach is applied in practice and the impact of each step, respectively. Figure 10 shows detailed memory optimization results (Figure 10a, as well as their corresponding impact on the execution time (Figure 10b). This detailed presentation of results allows the evaluation of the contribution of each step to the overall memory footprint and execution time reduction. The initial application memory footprint is measured at 258 MB, all statically allocated at the beginning of the execution. The execution time of the original version of the application for the specific input dataset is 1404.1s. A detailed analysis of the overall memory footprint requirements showed that the majority of allocated memory is used by imported modules. More specifically, 185MB were assigned to the imported libraries, while 69.1MB were allocated to store the input data and prediction models.

5.3.1. *Demonstration and impact of static to dynamic memory management*

After the application-level analysis (step 1), we implemented Step 2 of the methodology, to convert the most critical application data structures from static to dynamic and enable dynamic memory allocation and deallocation. Figures 11, 12 illustrate the comparison of the two approaches using a high-level application data-flow. As explained in Section 4, this step is manually implemented, as it requires an understanding of the application algorithm to avoid altering the application functionality. Originally, the prediction phase of the random forest algorithm was implemented statically, 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 prediction, as shown in Figure 11. 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. The RF classifier contains 20 prediction trees of 2.3MB each. Moreover, 18MB is used to store classifier metadata. Furthermore, in practice, not all classifier's information is required simultaneously. The required functionality and the necessary data can be provided to the algorithm dynamically when needed, only [11]. In other words, the dynamic loading of trees enables the dynamic allocation of memory required for the input data to each tree. More specifically, in the initial implementation, the test set and features memory size is 5.1MB. However, the transformation allowed the test to be split into 128 chunks of 0.41MB each, further reducing the RAM size requirements. Listing 1 shows a code snippet of how the source code is structured before(top) and after our transformation is applied(bottom). The modified version of the application is illustrated in Figure 12. This static-to-dynamic conversion reduced the overall memory footprint by 19.3% compared to the initial memory size, which corresponds to 50 MB, while enabling and increasing the impact of the optimizations of step 4 and step 5. Additionally, the execution time is reduced by 8.3%.

5.3.2. *Demonstration and impact of static and dynamic analysis*

The redundant data removal and code hierarchy reduction steps had also a significant impact on the memory size optimization of the application. Even though we propose a unified methodology, the automated proposed approach can operate independently to the non-automated steps. As we observe in Figure 10a, the optimizations based on static analysis reduced the memory requirements and the execution time by 34MB and 323.2s, respectively. The optimizations based on dynamic analysis reduced the required memory size by 56MB and the execution time by 273s. Aiming to provide further insights on the impact of the static and dynamic-based optimizations, we investigate how our application is transformed and what is the impact of each step, respectively.

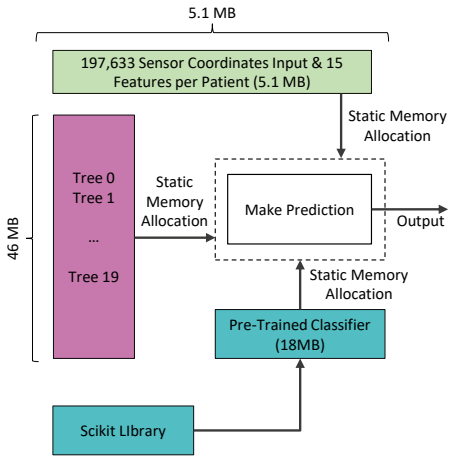


Figure 11: Initial static memory implementation of the IoT biomedical application.

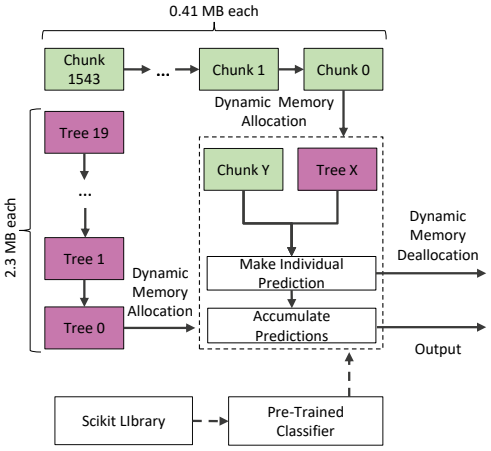


Figure 12: Dynamic memory implementation of the IoT biomedical application, after applying the optimization methodology.

Listing 2 depicts a code snippet of the redundant data removal. In this example, the function `estimateECGFearutres()` is never called through the source code, thus represented in the AST as leaf and is removed statically. On the other side, the `if` statement is input-dependent, thus cannot be removed statically, therefore is removed on dynamic analysis. Last but not least, the part of the code after the `return` function, which is outside the branch, can be removed only through dynamic analysis. This leads to fewer instructions invoked by the interpreter, while it resolves data dependencies for later steps.

As far as the code hierarchy reduction is concerned, this is implemented automatically through our framework. The application's version fed as input in this step is executed by importing 17 Python modules. Through redundant module removal, 23MB of memory is freed. Only the functionality of the utilized functions is moved to the main module, thus maintaining active only the necessary data. This leads to both less memory footprint and gains in execution time. Listing 3 illustrates a representative example of code hierarchy

reduction. For instance, the `estimateAR()` function is the only one utilized from the AR module, thus its functionality is transferred to the main module, while the rest of the AR is fully removed.

Furthermore, by exploiting the automated optimizations that Python's interpreter offers (step 6), the overall memory footprint reached 92.8MB. Compared to the original version of the application, the memory footprint reduction reached 64% (Figure 10a), while the execution time was reduced by 51.8%, as shown in Figure 10b. The execution time reduction is highly correlated to the memory footprint, due to the following reasons: i) the number of lines of code is significantly reduced, thus the interpreter's overhead and execution overhead is reduced, ii) the number of accesses to the main memory is reduced, thus CPU stalls are also reduced, iii) non-required memory allocations/deallocations and imports are avoided and iv) data and can fit within the cache, leading to more cache hits and faster execution times. Last but not least, Figure 13 illustrates the comparison of the initial and final memory utilization of IoT biomedical application. We observe that through our approach, the dynamic nature of the examined application can be exploited, thus leading to more effective utilization of memory. For low-resource devices, this dynamic behavior is critical for the efficient execution of an application.

5.4. Discussion

As shown in the above experiments, the proposed framework enables significant memory footprint optimizations, which also have a positive impact on execution time. For example, as can be noticed by Table 1 and Fig. 10a, the initial version IoT biomedical application could not be deployed on a Giant Board, on an Intel Galileo, or on a BeagleBone Black. However, after applying the proposed framework, the optimized version requires significantly less amount of resources and can be successfully executed.

```
# Static data load and processing
acc_data = pd.read_csv(acc_file) # Static load of data
classifier = open("all_trees.pckl") # Static load of RF trees
num_blocks = len(data['ACC'])/(4*32)
for i in range(num_blocks-1):   # Feed blocks to trees
    ... # Processing
-----------------------------------------------------------------
# Dynamic data load and processing
for gm_chunk in pd.read_csv(acc_file, chunksize=128,
     iterator=True): # Load data on chunks of 128 elements
    for i in range(0,20):
           # Load tree one by one
           pickle_in = open("tree"+str(i)+".pckl", "rb")
           ... # Processing
```

Listing 1: Code comparison before (top) and after (bottom) applying static to dynamic memory transformations step on IoT biomedical application

```
def estimateACCFeatures():
    ...
# Removed at static analysis as it is never called
def estimateECGFeatures():
    ...

class AR:
    def __init__(self, model_path, sensor_position,
         clf_method):
        ...
        # Removed at dynamic analysis as its execution is
             input dependent
        if clf_method is None:
            clf_method = 'rf'
        ...
        if sensor_position == 'chest':
            return
         # Rest of code removed at dynamic analysis as its
              execution depends on 'return' function trigger
        ...
```

Listing 2: Redundant code removal code snippet

To further investigate the impact of the proposed memory optimizations on other metrics, we monitored the energy consumption of the *pyperformance* benchmark and the IoT biomedical application, before and after applying the optimizations. The applications were deployed on an Nvidia Tegra X1 embedded device with 4 ARM Cortex-A57 processors running at 1.9 GHz and 4 GB RAM. The energy was measured by integrated power sensors plugged into the device. Figure 14 illustrates the energy gain percentage of the optimized *pyperformance* benchmarks, compared to the initial implementation. We observe that the energy consumption follows the behaviour of memory and performance optimizations [52], leading up to 55.1% less energy consumption and 19.3% on average. Similar experiments were conducted for the IoT biomedical application, showing 47.3% lower energy consumption.

Furthermore, an interesting question is to which extent the optimized Python implementation can compare to the corresponding C implementation in terms of execution time

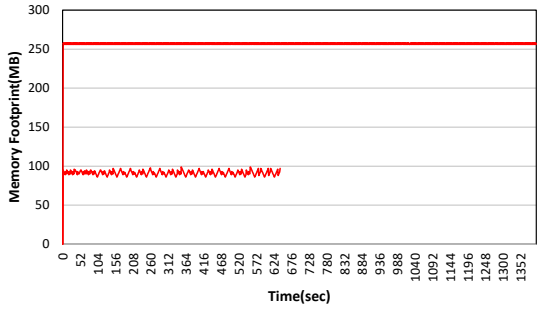


Figure 13: Comparison of initial and optimized memory footprint utilization

```
# Initial code hierarchy
... # Other imports
from AR import * # AR is initially imported
...
# Only estimateAR() is utilized from AR
current_ar = estimateAR(acc_data)
...
-----------------------------------------------------------------
# Reduced hierarchy
... # Other imports
# Only estimateAR() functionality is moved to main module
def estimateAR(self, df):
    ...
current_ar = estimateAR(acc_data)
```

Listing 3: Code comparison before (top) and after (bottom) applying code hierarchy reduction

and memory requirements. Therefore, the optimized version of the IoT application (i.e. the optimized Python source code after applying the proposed framework) was developed from scratch in C language. Both the Python and the C implementation of the IoT application were executed on an Nvidia Tegra X1. Table 4 summarizes the results of the comparison between Python and C implementation, using the same application input.

Regarding the execution time, the C implementation is 1.5x times faster, compared to the Python implementation. This is due to the fact that C is a compiler-based language, thus no code translation is required at run-time, and code optimizations can be applied by the compiler and further optimize the program using static information about types and memory layout of objects. In contrast to Python, which is an interpreter-based language, C avoids the run-time overhead of the interpreter and optimizations prior to execution cannot be achieved. This is a commonly observed behavior, as the identified overheads account for 64.9% of the overall execution time, while the remaining 35.1% is used for the execution of the program [19]. Therefore, there is at least 2.8x increase in execution time on average moving from a C-like program to a Python program running on CPython due to language and interpreter overheads. This observation is inline with corresponding results reported in the existing literature [30].

Additionally, we measured the number of total memory accesses of the corresponding applications by utilizing the *Linux perf* tool. More specifically, we monitored the performance counters of the processor throughout the execution of C and Python execution, respectively. We observe that Python requires an order of magnitude more memory accesses compared to the C implementation (23.8% more accesses). The underlying layers that CPython contains, induce a significant overhead for read and write accesses of the executed application.

Although the memory requirements of the Python implementation are higher and performance is lower than in C, it is friendlier to application developers. Indeed, as shown

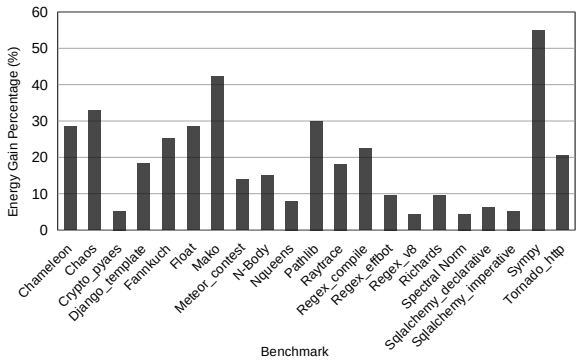


Figure 14: Energy consumption reduction percentage(%) for Pyperformance Benchmark Suite.

	Python Implementation	C Implementation
Memory Footprint	*92.8 MB*	*2.9 MB*
Total Memory Accesses	$31.4x10^6$	$74.9x10^5$
Execution Time	*677.25 sec*	*451.3 sec*
Lines of Code (LoC)	*1x*	*3.4x*

Table 4: Python and C code comparison

in Table 4 the C implementation of the IoT application requires 3.4x less LoC than the corresponding Python implementation. The fact that Python provides a variety of libraries and software abstractions makes it suitable for fast code development and contributes to its increased popularity in the embedded and HPC communities. The abstractions offered by Python can significantly contribute to highly maintainable source code, which is a critical feature for applications under development and for commercial software products. Even though it is probably not possible to achieve the same memory and execution time with corresponding C implementations, it is still important to reduce Python overheads as much as possible in an automated way, to allow developers to exploit Python's user-friendliness features in edge devices of limited resources.

6. Conclusion

In this work, we described a methodology supported by a tool flow for the memory optimization of Python applications targeting devices with limited computational resources at the edges of IoT networks. The methodology is enabled by dynamic memory management techniques and relies on two novel tools which automate the removal of redundant data and the reduction of the code hierarchy of Python applications. Although the optimizations focus on memory footprint reduction, we show that they have a significant positive impact on other metrics, including application execution time and energy consumption, which are also critical in the area of edge computing development.

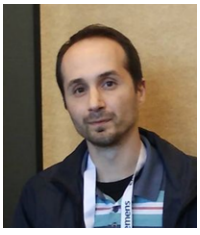
References

[1] M. Katsaragakis, L. Papadopoulos, C. Baloukas, D. Soudris, Memory management methodology for application data structure refinement

and placement on heterogeneous dram/nvm systems, in: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022, pp. 748–753.
[2] H. G. Abreha, M. Hayajneh, M. A. Serhani, Federated learning in edge computing: a systematic survey, Sensors 22 (2) (2022) 450.
[3] A. A. García, D. May, E. Nutting, Garbage collection for edge computing, in: 2020 IEEE/ACM Symposium on Edge Computing (SEC), IEEE, 2020, pp. 319–319.
[4] E. Oyekanlu, Predictive edge computing for time series of industrial iot and large scale critical infrastructure based on open-source software analytic of big data, in: 2017 IEEE International Conference on Big Data (Big Data), IEEE, 2017, pp. 1663–1669.
[5] M. M. Rahman, M. H. Sharker, R. Paudel, Active and collaborative learning based dynamic instructional approach in teaching introductory computer science course with python programming, in: 2020 IEEE Integrated STEM Education Conference (ISEC), IEEE, 2020, pp. 1–7.
[6] Pypl popularity of programming language (Feb. 2022).
URL https://pypl.github.io/PYPL.html
[7] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Ranking programming languages by energy efficiency, Science of Computer Programming 205 (2021) 102609.
[8] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Energy efficiency across programming languages: how do energy, time, and memory relate?, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, 2017, pp. 256–267.
[9] J. M. Redondo, F. Ortin, A comprehensive evaluation of common python implementations, IEEE Software 32 (4) (2014) 76–84.
[10] C. P. Lamprakos, L. Papadopoulos, F. Catthoor, D. Soudris, The impact of dynamic storage allocation on cpython execution time, memory footprint and energy consumption: An empirical study, in: International Conference on Embedded Computer Systems, Springer, 2022, pp. 219–234.
[11] M. Katsaragakis, L. Papadopoulos, M. Konijnenburg, F. Catthoor, D. Soudris, Memory footprint optimization techniques for machine learning applications in embedded systems, in: 2020 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2020, pp. 1–4.
[12] Micropython, https://micropython.org/ (Feb. 2022).
[13] A. Bartzas, S. Mamagkakis, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris, 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, IEEE, 2006, pp. 6–pp.
[14] T. Papastergiou, L. Papadopoulos, 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), IEEE, 2015, pp. 78–85.
[15] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quirós, A. Bartzas, F. Catthoor, D. Soudris, Dynamic memory management for embedded systems, Springer, 2015.
[16] A. Andrzejak, F. Eichler, M. Ghanavati, Detection of memory leaks in c/c++ code via machine learning, in: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2017, pp. 252–258.
[17] B. Powers, D. Tench, E. D. Berger, A. McGregor, Mesh: compacting memory management for c/c++ applications, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 333–346.
[18] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, C. Raffel, Learning-based memory allocation for c++ server workloads, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 541–556.
[19] M. Ismail, G. E. Suh, Quantitative overhead analysis for python, in: 2018 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2018, pp. 36–47.
[20] G. Barany, Python interpreter performance deconstructed, in: Proceedings of the Workshop on Dynamic Languages and Applications, 2014, pp. 1–9.
[21] N. Moshiri, Treeswift: A massively scalable python tree package, SoftwareX 11 (2020) 100436.
[22] A. R. Shajii, I. Numanagić, A. T. Leighton, H. Greenyer, S. Amarasinghe, B. Berger, A python-based optimization framework for high-performance genomics, bioRxiv.
[23] R. Power, A. Rubinsteyn, How fast can we make interpreted python?, arXiv preprint arXiv:1306.6047.
[24] T. W. Barr, R. Smith, S. Rixner, Design and implementation of an embedded python run-time system, in: 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), 2012, pp. 297–308.
[25] R. Monat, A. Ouadjaout, A. Miné, Value and allocation sensitivity in static python analyses, in: Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, 2020, pp. 8–13.
[26] M. Jump, K. S. McKinley, Cork: dynamic memory leak detection for garbage-collected languages, in: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2007, pp. 31–38.
[27] L. Cen, R. Marcus, H. Mao, J. Gottschlich, M. Alizadeh, T. Kraska, Learned garbage collection, in: Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2020, pp. 38–44.
[28] M. Ismail, G. E. Suh, Hardware-software co-optimization of memory management in dynamic languages, ACM SIGPLAN Notices 53 (5) (2018) 45–58.
[29] C. Baloukas, L. Papadopoulos, R. Pyka, D. Soudris, 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, 2010, pp. 155–160.
[30] B. Ilbeyi, C. F. Bolz-Tereick, C. Batten, Cross-layer workload characterization of meta-tracing jit vms, in: 2017 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2017, pp. 97–107.
[31] J. Chou, L. Chen, H. Ding, J. Tu, B. Xu, A method of optimizing django based on greedy strategy, in: 2013 10th web information system and application conference, IEEE, 2013, pp. 176–179.
[32] https://www.python.org/ (April 2022).
[33] Finalizers, https://docs.python.org/2/c-api/init.htmlc.Py_Finalize (April 2023).
[34] https://docs.python.org/3/c-api/memory.html.
[35] https://realpython.com/python-memory-management/#cpythons-memory-management.
[36] https://rushter.com/blog/python-memory-managment/.
[37] https://github.com/pythonprofilers/memory_profiler.
[38] https://github.com/zhuyifei1999/guppy3/.
[39] https://pypi.org/project/pyflakes/.
[40] https://pypi.org/project/autoflake/.
[41] https://coverage.readthedocs.io/en/v4.5.x/.
[42] https://docs.python.org/3/library/trace.html.
[43] H. Zhai, C. Casalnuovo, P. Devanbu, Test coverage in python programs, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 116–120.
[44] S. Vappangi, N. K. Penjarla, S. E. Mathe, H. K. Kondaveeti, Applications of raspberry pi in bio-technology: A review, in: 2022 2nd International Conference on Artificial Intelligence and Signal Processing (AISP), IEEE, 2022, pp. 1–6.
[45] A. C. De Melo, The new linux'perf'tools, in: Slides from Linux Kongress, Vol. 18, 2010, pp. 1–42.
[46] Pyperformance, https://pyperformance.readthedocs.io/.
[47] https://github.com/OmarAflak/Medium-Python-Neural-Network (April 2023).
[48] G. Cohen, S. Afshar, J. Tapson, A. Van Schaik, Emnist: Extending mnist to handwritten letters, in: 2017 international joint conference on neural networks (IJCNN), IEEE, 2017, pp. 2921–2926.

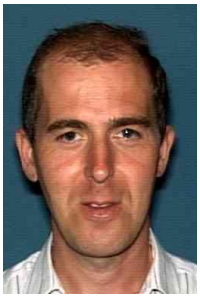
[49] A. Liaw, M. Wiener, et al., Classification and regression by randomforest, R news 2 (3) (2002) 18–22.
[50] Imec, https://www.imec-int.com/en (Feb. 2022).
[51] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, et al., Sympy: symbolic computing in python, PeerJ Computer Science 3 (2017) e103.
[52] M. Katsaragakis, C. Baloukas, L. Papadopoulos, V. Kantere, F. Catthoor, D. Soudris, Energy consumption evaluation of optane dc persistent memory for indexing data structures, in: 2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC), IEEE, 2022, pp. 75–84.

Manolis Katsaragakis received the Diploma degree in electrical and computer engineering from the National and Technical University of Athens (NTUA), Greece. Currently, he is a joint-PhD student with the microprocessors and digital systems laboratory of electrical and computer engineering school at NTUA and in Katholieke Universiteit Leuven (KUL), Belgium in cooperation with IMEC, Belgium. His main research interests include dynamic memory management techniques, both in embedded systems and HPC domains. He has also worked for EU H2020 project EXA2PRO and NEPHELE.



Lazaros Papadopoulos received the Diploma degree in electrical and computer engineering from the Democritus University of Thrace, Greece, and the Ph.D. degree from the School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece. He is a Research Associate with the Microprocessors and Digital Systems Laboratory, National Technical University of Athens. His current research interests include memory management optimization techniques for embedded systems and power aware computing.

Mario Konijnenburg (M'08) received an M.S. degree in electrical engineering from Delft University of Technology in The Netherlands in 1993. A Ph.D. degree was received from Delft University of Technology in 1999 on Automatic Test Pattern Generation for Sequential Circuits. He joined Philips Research/NXP Semiconductors and worked on methodologies to improve testability of designs. Currently, he is chip architect and R&D manager of the IC-design group at imec in Eindhoven, The Netherlands, targeting chip research for bio medical applications covering SoC design, sensors, stimulators, power, and (RF) communication.



Francky Catthoor received the PhD degree from the Katholieke Universiteit Leuven, Belgium, in 1987. He is a fellow at the Interuniversity Microelectronics Center (IMEC), Heverlee, Belgium. His current research activities include architecture design methods and system-level exploration for power and memory footprint within real-time constraints, oriented toward data storage management, global data transfer optimization, and concurrency exploitation, also including deep-submicron technology issues. He was elected an IEEE fellow in 2005. He is also a member of the IEEE Computer Society.

Dimitrios Soudris received the diploma and PhD degrees in electrical engineering from the University of Patras, Greece, in 1987 and 1992, respectively. Since 1995, he has been a professor with the Department of Electrical and Computer Engineering(ECE), Democritus University of Thrace, Xanthi, Greece for thirteen years. He is a professor with the School of ECE, NTUA, Greece. He has (co)authored more than 500 papers in international journals/conferences and coauthored/-coedited eight Kluwer/Springer books. He is the leader and a principal investigator in a plethora of research projects funded by the Greek Government and Industry, European Commission, ENIAC-JU, and European Space Agency. His current research interests include HPC, embedded systems, reconfigurable architectures, reliability, and low-power VLSI design. He was a recipient of the Award from INTEL and IBM for the EU project LPGD 25256 and the ASP-DAC 05 and VLSI 05 awards for EU AMDREL IST-2001-34379, as well as several HiPEAC awards.