Pygration : Workload-Aware Live Migratable Cloud Instance Detector for Python Runtime

Soohyuk Lee, Junho Lim, Kyungyong Lee, *Member, IEEE*

***Abstract*—Supporting live migration in the cloud can be beneficial to dynamically build a reliable and cost-optimal environment, especially when using spot instances. When a spot instance interruption event occurs, users can apply live migration using the Checkpoint/Restore In Userspace (CRIU) to a more reliable instance. In the process of migration, ensuring the compatibility of the CPU features between the source and target hosts is crucial for flawless execution after migration. However, the standard approach implemented by CRIU is workload-agnostic and overly conservative, comparing the full CPU feature sets of the hosts, which results in unnecessarily restricting the pool of viable migration targets. To mitigate this limitation, a workload-aware analysis can be employed to identify the precise set of CPU features that an application utilizes at runtime. However, it can be very challenging for Python workloads due to their layers of abstraction between bytecode and invoked native libraries, obscuring the true hardware dependencies. To overcome the challenge, this paper presents Pygration, a novel workload-aware migratable cloud instance detection system for Python runtimes. Pygration implements a hybrid analysis pipeline that combines Python bytecode tracking to build a precise call graph with a native code execution path tracking heuristic to identify the minimal set of required CPU features. A comprehensive evaluation on 522 AWS instance types shows that Pygration achieves perfect precision while improving recall by over 5× compared to the CRIU baseline. In a practical spot instance scenario, this increased recall translates to a 16% improvement in median cost savings while enhancing reliability.**

***Index Terms*—Migration, ISA, Python, Compatibility, Cloud**

I. Introduction

THE evolution of cloud computing has fundamentally changed how computing resources are utilized. Key characteristics such as elastic resource provisioning, on-demand billing, and a vast variety of instance types have led to its broad adoption. This diversity of instances presents a significant opportunity to enhance operational efficiency through live migration, which allows applications to be moved to more suitable hardware as their resource requirements change. At the process or container level, this can be performed without direct support from cloud providers by using user-space migration systems, such as CRIU [1]. A critical requirement for migrating an application between instances with distinct CPU models or generations is guaranteeing its correct operation post-migration. To ensure safety, the destination host must support all CPU features utilized by the workload. The standard approach, implemented by CRIU's compatibility checker, is to conservatively verify that the target instance supports the full set of CPU features advertised by the source. However, this workload-agnostic strategy has a significant drawback: an application typically uses only a small subset of the features available on its host. By considering unused features, this method unnecessarily disqualifies a large number of viable targets, a problem that can be characterized as a high rate of false-negative detections, resulting in low recall. This false-negative migratable instance detection challenge is amplified for applications written in high-level interpreted languages like Python. The Python execution model introduces layers of abstraction that obscure the underlying hardware interactions. Code is compiled into platform-independent bytecode and executed by the Python Virtual Machine (PVM), which delegates performance-critical operations to precompiled native libraries via a Foreign Function Interface (FFI). The actual hardware-specific instructions (e.g., AVX512) are contained within these native libraries, making their dependencies opaque to an analysis of the PVM or bytecode alone. Therefore, a reliable compatibility check for Python workloads demands a holistic approach that can analyze both the PVM's baseline requirements and the specific native code paths invoked by the application. To address this challenge, this paper presents Pygration, a workload-aware compatibility detector designed specifically for Python applications. Pygration implements a novel, multi-stage analysis pipeline that bridges the gap between high-level Python code and the low-level native instructions it executes. Its core methodology combines Bytecode Tracking, which simulates the PVM stack to build a precise call graph of the Python application, with Execution Path Tracking, a heuristic that traces only the reachable code paths within the invoked native libraries. This hybrid approach allows Pygration to construct a minimal and accurate set of required CPU features for any given Python workload. A comprehensive evaluation of Pygration was conducted using a testbed of 522 unique AWS EC2 instance types across 14 diverse Python workloads. The experimental results demonstrate that the proposed system identifies compatible migration targets with a perfect precision of 1.0. Compared to the conservative baseline approach of CRIU, Pygration achieves over a 5× improvement in recall, significantly expanding the pool of viable migration instances. The practical efficacy of this higher recall was demonstrated in a spot instance environment, where Pygration improves

Soohyuk Lee is with the Department of Computer Science, Kookmin University, 77, Jeongneung-ro, Seongbuk-gu, Seoul, Republic of Korea. E-mail: cg10036@kookmin.ac.kr

Junho Lim is with PIOLINK, 98, Gasan digital 2-ro, Geumcheon-gu, Seoul, Republic of Korea. This work was done when he was a student at Kookmin University. E-mail: lim-junho@kookmin.ac.kr

Kyungyong Lee is with the Department of Data Science, Hanyang University, 222, Wangsimni-ro, Seongdong-gu, Seoul, Republic of Korea. E-mail: kyungyong@hanyang.ac.kr (Corresponding author: Kyungyong Lee)

median cost savings by 16% while simultaneously enhancing the reliability and availability of the selected instances. The primary contributions of this work are as follows:

- A novel, workload-aware system, Pygration, that accurately identifies migratable instances for Python applications by integrating bytecode analysis with native code execution path tracing.
- A robust implementation that addresses the complexities of Python's execution model, including sophisticated symbol resolution for mapping high-level code to its underlying native library implementations.
- A comprehensive evaluation demonstrating that Pygration significantly improves migration recall over baseline methods, leading to tangible improvements in reliability and cost savings in real-world cloud environments.

II. Runtime Migration on Public Cloud

Runtime migration in public cloud environments refers to the technique of dynamically transferring a running application from one host machine to another. A particularly compelling form of this is live migration, which preserves the entire state of an application, including its volatile memory content, open file descriptors, and network sockets, to ensure seamless execution continuity. Migration can be applied to achieve load balancing of computational resources [2], [3] to serve Large Language Model (LLM) inference [4], improving power efficiency [5], [6], and ensuring continuous operation even in disaster scenarios [7]. In a public cloud computing service, hypervisor-level live migration of virtual machines (VMs) [8] is a feature typically reserved for internal use by service providers for infrastructure maintenance, but users can initiate migration at finer granularity, such as for containers [2], [9] or individual processes [1], [10] operating within a VM instance.

A. Live Migration Process

For user-space live migration of processes and containers, CRIU has emerged as a standard tool. It facilitates the migration of an application while preserving its complete runtime state, thereby preventing data loss or significant downtime. The objects encompassed by a CRIU migration include virtual memory mappings, CPU register contents, file descriptors, network sockets, and the process tree structure (pstree). The CRIU migration procedure is divided into two primary phases: checkpointing and restoration.

a) Checkpointing Phase: To begin, the target process is suspended using the cgroup freezer feature, which prevents any state changes during the serialization process. CRIU then recursively scans the /proc/$pid/ directory to enumerate all resources associated with the process. Subsequently, it employs the ptrace system call to inject a parasite code segment into the address space of the target process. This allows CRIU to execute within the process's context to dump its memory contents and other state information. The resulting checkpoint, serialized in the Protocol Buffers (protobuf) [11] format, is stored for subsequent restoration.

b) Restoration Phase: During restoration, CRIU first parses the checkpoint files to identify all resources, including shared ones like file handles and network connections. It then invokes the fork system call multiple times to reconstruct the original process tree and prepare the necessary address spaces. Finally, it restores the application's state, including memory mappings, file descriptors, and socket states, based on the information captured in the checkpoint files, allowing the process to resume execution from its pre-checkpointed state.

B. Live Migration Between Cloud Spot Instances

To capitalize on surplus computational capacity, public cloud providers offer underutilized resources at a significant discount, commonly referred to as spot instances [12], [13]. The primary trade-off for this cost reduction is the volatile nature of spot instances; they can be preemptively terminated by the provider with minimal warning when the capacity is required for on-demand users. This inherent unreliability poses a significant challenge for running stateful or long-running applications. In the literature, a considerable body of research has been conducted to address spot instance interruptions. These strategies include setting a proper spot instance bidding price [13], [14], using the checkpointing feature inherent to a specific application to restart from a checkpointed state when an interruption occurs [12], [15], or utilizing a mixture of spot and on-demand instances to achieve a balance between cost efficiency and reliability [16], [17]. In contrast to these methods, live migration presents a more general strategy to mitigate the impact of such interruptions. This approach preserves service continuity by checkpointing an application on a spot instance facing termination and restoring it on a more stable host. Such a host could be another spot instance with a lower interruption risk or a standard on-demand instance. This approach is particularly valuable for applications that do not support native checkpointing capabilities, thereby expanding the range of workloads that can leverage the economic benefits of spot instances [18]. The ability to dynamically relocate workloads allows users to achieve a balance between cost efficiency and operational reliability [16]. This strategy, however, requires identification of a compatible target instance for migration. The selection is non-trivial, as the cost savings and underlying hardware characteristics vary significantly across the hundreds of available instance types [19]. Therefore, developing a reliable mechanism to detect compatible migration targets emerges as a critical challenge.

C. Ensuring Compatibility Between Cloud Instances

A critical requirement for reliable live migration is that the destination host must support all CPU features actually utilized by the migrating process. If the restored process attempts to execute an instruction that is not supported by the target CPU, the system raises an invalid opcode signal (SIGILL), resulting in immediate termination. To prevent such failures, CRIU [1] adopts a conservative strategy: a process is considered migratable only if the destination host supports the full set of CPU features advertised by the source host. While this approach guarantees correctness, it often leads to a high

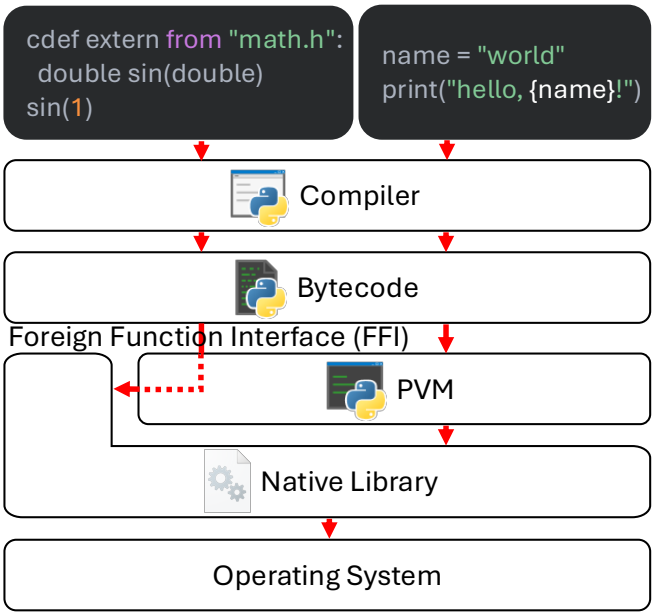


Fig. 1. The execution sequence of the Python program

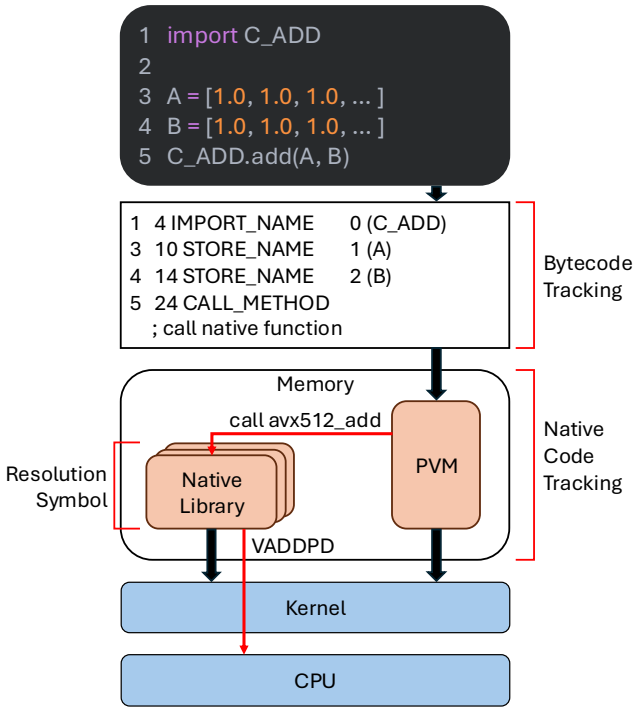


Fig. 2. Modules of proposed Python workload execution tracking system composed of PVM tracking for native libraries analysis, bytecode tracking, and symbol resolution between bytecode and native library.

rate of false-negative detections by unnecessarily excluding compatible hosts.

1) Verifying Native Binary Workload Compatibility: The limitations of CPU feature-based compatibility checks can be addressed through workload-aware analysis that identifies the exact CPU features exercised by an application at runtime. For natively compiled applications, this involves statically analyzing the executable binary and its dynamically linked libraries to construct a precise feature usage profile [18]. This fine-grained approach significantly broadens the set of eligible migration targets for native binary workloads while preserving correctness and safety. By targeting only the CPU features actually used during execution, it avoids overly conservative decisions and supports more flexible, cost-effective migration in heterogeneous cloud environments.

2) Challenges in Compatibility Checking for Python Workloads: While workload-aware analysis has proven to be effective for natively compiled applications [18], it presents unique challenges when applied to interpreted languages, such as Python, whose execution model introduces layers of abstraction that conceal the underlying hardware interactions required for accurate compatibility analysis. The process of Python code execution flow is shown in Figure 1. When a Python script (`.py`) is executed, it is first compiled into a platform-independent intermediate form known as *bytecode*. This bytecode is then interpreted by the PVM, rather than executed directly by the host operating system. While this abstraction ensures cross-platform portability, it complicates compatibility analysis by hiding low-level hardware interactions. Moreover, Python's interpreted nature introduces performance overhead, especially for compute-intensive workloads. To address these limitations, particularly in domains like scientific computing and machine learning, Python commonly delegates computationally intensive operations to native libraries written in languages such as C or C++. These libraries are invoked through a Foreign Function Interface (FFI), allowing Python code to seamlessly call precompiled native functions. For example, a high-level API call in TensorFlow [20] may internally trigger a native function optimized with instruction sets like *AVX512_VNNI* to expedite the execution of convolutional neural network (CNN) operations. From the Python code's perspective, these lower-level interactions remain entirely abstracted. This layered architecture presents two critical migration compatibility check challenges. First, the PVM itself, as a native binary, requires its own CPU compatibility verification. Second, and more critically, the native libraries invoked via the FFI have hardware dependencies that are entirely opaque to an analysis of the interpreter (PVM) alone. Therefore, ensuring the safe migration of a Python workload demands a comprehensive approach capable of analyzing through these layers of abstraction. The strategy must involve analyzing the Python bytecode to trace potential FFI calls to their native libraries and build a complete profile of all CPU features required by a Python workload.

III. Tracking Python Workload Execution

To track the actual CPU features utilized in a Python program, we propose a multi-stage analysis pipeline designed to bridge the gap between high-level Python code and the low-level CPU instructions it executes. As illustrated in Figure 2, this pipeline is decomposed into two parallel tracking phases followed by a final resolution phase. The first phase, **Native Library Tracking** (Section III-A), examines the executable instructions loaded into the process's `.text` memory segment. This analysis determines the specific CPU features utilized by

both the PVM itself and all loaded native library functions. Concurrently, the **Bytecode Tracking** (Section III-B) module performs a static analysis of the Python bytecode. It constructs a call graph by parsing the bytecode to identify all potential function and method invocations, with a particular focus on pinpointing calls directed to external native libraries. The primary challenge lies in accurately mapping the symbolic function names identified during bytecode tracking to the real implementations found in memory by the native library tracking. A high-level function call in Python source code may not directly correspond to its symbol in a compiled binary; for instance, a call to *add* might be linked to a highly optimized native function like *avx512_add*, as shown in Figure 2. The final **Symbol Resolution** phase (Section III-C) is designed to solve this discrepancy. It correlates the symbolic function calls from the Python code level with the actual operations invoked by native libraries, creating a precise link between them. By integrating these phases, the proposed system can map a high-level Python operation to the specific CPU features that its underlying native implementation executes. This allows for the construction of a comprehensive and accurate required CPU feature map for the entire Python workload, enabling precise compatibility evaluation for migration.

A. Tracking PVM Execution Path via Native Library Analysis

The PVM, as the core execution engine for Python programs, is a native application typically implemented in C and linked against native system libraries. As such, it is subject to the same hardware compatibility constraints as any other compiled binary. Verifying its compatibility across heterogeneous cloud instance types requires analyzing its potential execution paths to identify the specific CPU features it utilizes. This section details two heuristics to perform this analysis: the *text-segment full scan* and *execution path tracking*. Both methods operate by inspecting the instructions within the process's memory (i.e., the `.text` segment) and utilizing the Intel X86 Encoder Decoder (XED) [21] to map these instructions to their corresponding CPU features.

1) Text Segment Full Scan: The text-segment full scan is a heuristic that exhaustively analyzes every executable instruction within a process's memory space. The implementation utilizes the GNU Debugger (GDB) to identify the memory addresses of the `.text` segments belonging to the main process and all loaded shared libraries. The machine code within these segments is then disassembled into assembly instructions using Capstone disassembler [22]. Finally, each unique instruction is decoded using the Intel X86 XED [21] to map it to its corresponding CPU feature requirement. As detailed in Algorithm 1, the procedure begins by identifying all text sections of the target process and its libraries (Line 4). It then traverses these memory regions, disassembles the machine code, and aggregates all unique instructions into a single set (Lines 5-6). Finally, the algorithm iterates through this set of unique instructions, decodes each one to find its associated CPU feature, and compiles the final set of all required features (Lines 8-11). The primary advantage of this heuristic is its completeness; it guarantees the identification of every possible instruction the program might execute. However, this exhaustive approach has significant drawbacks. It incurs high analysis overhead due to the large volume of code being disassembled and decoded. More importantly, it often results in false-negative detections for migration compatibility. For example, the GLIBC function `strlen` has multiple versions optimized for different instruction sets (e.g., AVX2, AVX512). The full-scan method identifies all these versions as execution candidates, even though only one will be used at runtime, thus unnecessarily restricting the pool of viable migration targets.

Algorithm 1: Text Segment Full Scan

Input: P: A target process
Output: F_{req}: A set of required CPU features

1 $I_{all} \leftarrow$ new Set()
2 $F_{req} \leftarrow$ new Set()
3 $T_{sections} \leftarrow$ get_text_sections(P)
4 **for** *each* T ***in*** $T_{sections}$ **do**
5 $\quad I_{current} \leftarrow$ disassemble(T.start_addr, T.end_addr)
6 $\quad I_{all}$.update($I_{current}$)
7 **end**
8 **for** *each* I ***in*** I_{all} **do**
9 $\quad f \leftarrow$ xed_decode(I)
10 $\quad F_{req}$.add(f)
11 **end**
12 **return** F_{req}

2) Execution Path Tracking: To overcome the high overhead and false-negative rate of the full-scan method, we propose the *Execution Path Tracking* (EPT) heuristic. Instead of analyzing all instructions in memory, this approach identifies the CPU features in use by statically tracing the program's call graph from its entry point. This significantly reduces the analysis scope to only the code paths that are potentially reachable during execution. However, accurately tracing these paths is non-trivial, as function calls can be resolved through several mechanisms, as illustrated in Figure 3. A *Direct call* resolves to a fixed memory address known at compile time. In contrast, calls to dynamically linked library functions are resolved at runtime via the Procedure Linkage Table (PLT), which acts as an intermediary to look up the function's address in the Global Offset Table (GOT) upon first use (*Through PLT*). Modern compilers may also employ a *Bypass PLT*, where calls directly reference an address in the GOT that was resolved at program startup. An effective path-tracking algorithm must be able to correctly identify the target of a call in all three scenarios. The implementation of native library execution path tracking, detailed in Algorithm 2, performs a recursive traversal of the program's call graph. The process begins by placing the program's main entry point onto a queue (Line 2). The algorithm then repeatedly dequeues a function, disassembles its machine code, and adds its unique instructions to a global set, I_{all} (Lines 4-10). For each disassembled function, the algorithm identifies all branch instructions (e.g., `call`, `jmp`) and attempts to resolve their target addresses (Lines 12-15). To handle the various call mechanisms, the

```c
// module.c
#include "module.h"

int sub(int a, int b) { return a - b; }
```

```c
// main.c
#include "module.h"

int add(int a, int b) { return a + b; }

int main() {
    // Direct call
    int result1 = add(3, 4);
    // Call sub in module through PLT
    int result2 = sub(3, 4);

    return 0;
}
```

```c
// got-example.c
// gcc got-example.c -o got-example -fno-plt
#include "module.h"

int main() {
    // Call sub in module Bypass PLT
    int result = sub(3, 4);
    return 0;
}
```

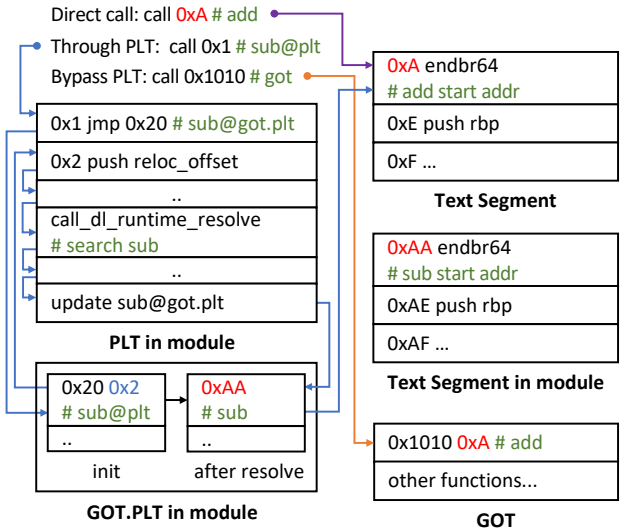


Fig. 3. Different methods of calling functions of direct call, through PLT, and bypass PLT

implementation leverages metadata from debugging tools like GDB. The annotated disassembly provided by GDB often reveals the true destination of a branch, whether it is a direct address or a symbol resolved through the PLT and GOT. If a target is successfully resolved and has not been visited, it is added to the queue for subsequent analysis (Line 14). After the traversal is complete, all unique instructions collected are decoded to produce the final set of required CPU features (Line 18). The EPT approach offers a significant advantage over the full-scan method. It naturally handles compiler optimizations like function multi-versioning [23], as only the specific version of a function (e.g., an AVX2-optimized `strlen`) is on the actual execution path chosen at startup. This leads to a more

Algorithm 2: Execution Path Tracking

Input: P: A target process
Output: F_{req}: A set of required CPU features

1 $Q \leftarrow$ new Queue(); $V \leftarrow$ new Set(); $I_{all} \leftarrow$ new Set();
2 Q.enqueue(get_entry_point(P))
3 **while** Q *is not empty* **do**
4 $\quad F_{current} \leftarrow Q$.dequeue()
5 $\quad$ **if** $F_{current}$ ***in*** V **then**
6 $\quad\quad$ **continue**
7 $\quad$ **end**
8 $\quad V$.add($F_{current}$)
9 $\quad I_{current} \leftarrow$ disassemble($F_{current}$)
10 $\quad I_{all}$.update($I_{current}$)
11 $\quad$ **for** *each branch instruction* I ***in*** $I_{current}$ **do**
12 $\quad\quad F_{target} \leftarrow$ resolve_target(I)
13 $\quad\quad$ **if** F_{target} *is not null* **then**
14 $\quad\quad\quad Q$.enqueue(F_{target})
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 **end**
18 $F_{req} \leftarrow$ decode_features(I_{all})
19 **return** F_{req}

precise set of required features and fewer false negatives. The primary limitation of this static analysis, however, is its inability to resolve function calls made via pointers or register values (e.g., `call eax`), as their targets are determined dynamically at runtime.

B. Tracking Python Bytecode

To determine the full extent of a Python workload's hardware dependencies, it is necessary to construct a call graph of Python source code that maps all potential execution paths, including those into native libraries. Since these dependencies are not visible in the source code, our approach requires a direct analysis of Python's intermediate representation, which is referred to as the bytecode. As illustrated in Figure 4, disassembled bytecode provides a structured, sequential view of a program's operations. Each instruction consists of four main components: the corresponding source code line number, its offset in the bytecode sequence, the operation code (opcode) itself, and any arguments the opcode requires. Understanding this structure is the first step toward tracing the program's execution flow. An intuitive approach to building a call graph would be to simply identify all `CALL_*` instructions within the bytecode. However, this method is insufficient. A `CALL` instruction in Python bytecode does not contain the name of its target function; it merely executes whichever callable object is currently on top of the PVM stack. This presents a challenge, as the identity of the called function can only be determined by understanding the stack's state at the moment of the call. For instance, the `CALL_METHOD` for `time.sleep(1)` in the example is only meaningful after preceding `LOAD` instructions have pushed the `time` module, the `sleep` method, and its

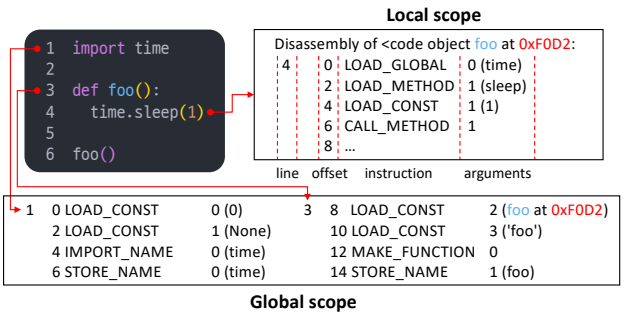


Fig. 4. An example of how Python code is expressed using a bytecode format

Algorithm 3: Bytecode-based Call Graph Construction

Input: B: Bytecode instructions for a function, $F_{current}$
Output: G: A call graph

```
1  S ← new Stack(); G ← new Graph() for ins in B do
2      get_instruction_family(ins.opcode)
3      case IMPORT do
4          module ← resolve_import(ins)
5          S.push(module)
6      end
7      case LOAD do
8          operand ← resolve_operand(ins)
9          S.push(operand)
10     end
11     case STORE or POP do
12         S.pop()
13     end
14     case CALL do
15         num_args ← ins.arg
16         args ← S.pop_n(num_args)
17         callable ← S.pop()
18         G.add_edge(F_current, callable)
19     end
20 end
21 return G
```

arguments onto the stack. Algorithm 3 presents the static analysis procedure for constructing a call graph from Python bytecode. The algorithm operates by traversing the instruction sequence of a given function while simulating the behavior of the PVM's stack machine. To achieve this, a virtual stack (Line 1) is maintained to model the operand stack's state at each instruction offset. During this traversal (Line 2), the state of the virtual stack is updated according to the semantics of each opcode. Instructions such as `IMPORT_*` and `LOAD_*` result in object representations being pushed onto the stack (Lines 4, 8), while `STORE` and `POP` instructions remove an item (Line 13). Conversely, call resolution is triggered by a `CALL_*` instruction (Line 15). The target callable and its arguments are retrieved by popping from the stack based on the instruction's argument count (Lines 16-18), and a new directed edge is added to the graph G (Line 19). This process, applied recursively to functions within the workload, yields a comprehensive call graph that maps all potential execution paths, including those into native libraries.

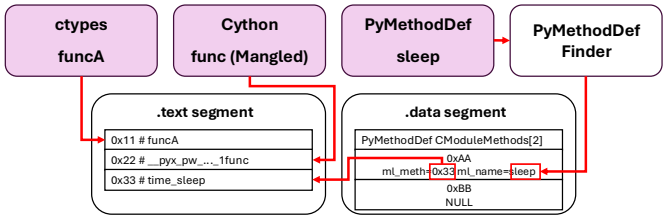


Fig. 5. Different ways of resolving Bytecode tracking function to the native library function

C. Tracking Native Library Call From Bytecode

The proposed analysis framework integrates two complementary tracking heuristics: *Execution Path Tracking* for native binaries, including PVM, and *Bytecode Tracking* for Python code. The pipeline is designed such that symbolic native function calls identified by the Bytecode Tracker must be subsequently traced by the Execution Path Tracking module to determine their CPU feature requirements. However, a key complexity arises in bridging these two stages: the symbolic name of a function at the Python level often does not directly correspond to its exported symbol in the compiled native library. For instance, a Python wrapper may expose a function as `foo`, while its underlying C implementation is named `_Py_foo_internal` for linkage purposes. To bridge this gap, the resolution strategy must be tailored to the specific methods, and this section details the three primary mechanisms we address, as illustrated in Figure 5.

a) Direct Symbol Mapping with `ctypes`: For native libraries loaded using Python's `ctypes` library [24], symbol resolution is straightforward. `ctypes` loads standard shared libraries (.so, .dll) and accesses their functions without altering the symbol names. Therefore, the function name identified from the bytecode can be used directly to look up its address within the process's `.text` section.

b) Demangling Cython Symbols: When a library is built using Cython [25], the compiler performs name mangling to avoid naming conflicts and encode type information. This process transforms the Python-level function name into a unique C-level symbol. For instance, a function `module.submodule.func` is compiled into a native function with a mangled name such as `__pyx_pw_6module_9submodule_1func`. To resolve these calls, our system applies Cython's known mangling rules to the symbolic name before searching for the resulting mangled symbol in the native binary.

c) Table-based Resolution for the Python C API: For modules built with the standard Python C API, such as many of CPython's own built-in functions, linkage is managed through a static mapping table (the `PyMethodDef` structure). This structure, typically located in the library's `.data` section, explicitly links the public-facing function name (e.g., "sleep") to a function pointer containing the memory address of the actual C implementation (e.g., `&time_sleep`). While this

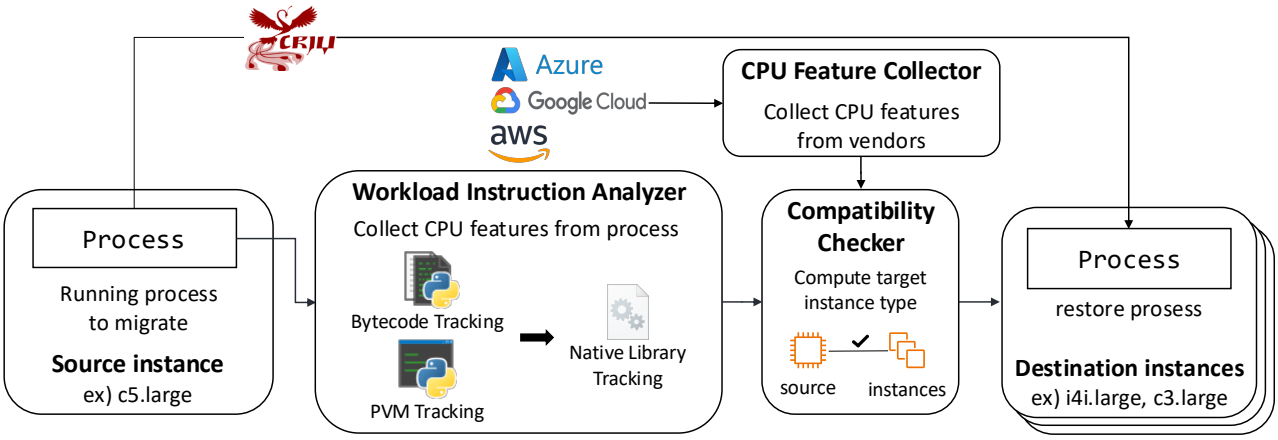


Fig. 6. Components of the proposed system to find a set of cloud instances to which can migrate

Algorithm 4: Python C API Symbol Resolution

Input: L: A native library file
S_{target}: A target function symbol string
Output: A_{impl}: The address of the native function implementation, or `null`

```
1  D_sec ← get_data_section(L)
2  V_structs ← find_candidate_structs(D_sec)
3  M_defs ← new List()
4  for each v in V_structs do
5      if matches_PyMethodDef_signature(v) then
6          M_defs.append(v)
7      end
8  end
9  for each pmd in M_defs do
10     s_name ← read_string_at(pmd.ml_name)
11     if s_name == S_target then
12         return pmd.ml_meth
13     end
14 end
15 return null
```

table can be located directly if debugging symbols are present, these symbols are often stripped in production environments to reduce library size. To overcome this, the proposed system employs the heuristic procedure detailed in Algorithm 4. The process begins by identifying all structure-like global variables within the library's `.data` section (Lines 1-2). Each candidate structure is then validated against the known memory signature of a `PyMethodDef` struct to verify its field count, data types, and sizes to filter probable mapping tables (Lines 4-6). Once the set of likely `PyMethodDef` structures is identified, the algorithm iterates through them to find the target function (Lines 9-14). For each structure, it reads the function name string from the `ml_name` field and compares it against the target symbol. If a match is found, the function pointer from the `ml_meth` field is returned, which is the memory address of the actual native implementation. This address can then be used to trace the function's execution path within the `.text` section.

IV. Migratable Cloud Instance Detector

Conventional compatibility checkers implemented in CRIU are overly conservative by checking the entire CPU feature compatibility between instances. To address this, we propose Pygration, a workload-aware system designed to accurately identify migratable cloud instances for Python applications. Pygration achieves high precision and recall by implementing distinct analysis modules that trace dependencies through both Python bytecode and the native libraries they invoke. Figure 6 illustrates the architecture of Pygration, which is composed of three core components that collaborate to identify compatible migration targets. The *CPU Feature Collector* operates offline, gathering the specific CPU features of diverse cloud instance types. The *Workload Instruction Analyzer* is the core online component. When a migration is needed, it performs the deep analysis detailed in the preceding sections of tracing both the PVM and its native library dependencies to produce a precise set of required CPU features for that specific workload. The *Compatibility Checker* takes the required feature set from the analyzer and compares it against the database of available features of various cloud instances from the collector. Its output is a definitive list of cloud instances to which the application can be safely migrated.

A. CPU Feature Collector

To enable rapid compatibility decisions, the *CPU Feature Collector* module gathers the hardware capabilities of various cloud instance types in an offline manner. Given that public cloud providers offer hundreds of unique instance types with distinct CPU features, this pre-collected database is essential for on-demand migration analysis. The method for extracting these features varies by architecture; x86 systems utilize the `CPUID` instruction, while ARM systems rely on specific system registers. As the majority of cloud instances are based on the x86-64 Instruction Set Architecture (ISA), this work focuses on compatibility within this ecosystem. The collector's implementation for the x86 architecture leverages the `__get_cpuid_count` function provided by the GNU C

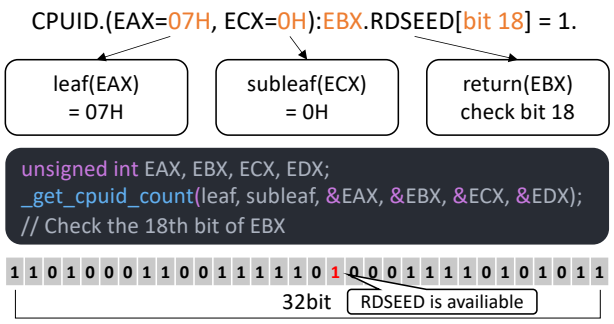


Fig. 7. Using the CPUID instruction to check for host support of a specific CPU feature.

Library (glibc) to execute the `CPUID` instruction. The behavior of this instruction is controlled by input values, known as *leaves* and *subleaves*, passed via the `EAX` and `ECX` registers, respectively. The resulting data, which details the CPU's model, family, and supported features, is returned across several registers. For example, to verify support for the Read Random Seed (`RDSEED`) feature, the collector queries the specific leaf defined in the architecture manual: `CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18]`. As illustrated in Figure 7, this is achieved by invoking `__get_cpuid_count` with `EAX` set to `07H` and `ECX` to `0H`. The feature's availability is then confirmed by checking if the 18th bit of the returned `EBX` register is set.

B. Workload Instruction Analyzer

The *Workload Instruction Analyzer* is the central component of Pygration, responsible for orchestrating the various analysis techniques to produce a comprehensive set of CPU feature requirements for a given Python workload. This process involves the integration of the three specialized modules detailed in the preceding sections. The analysis begins with two parallel tracks. First, the native library tracking is applied directly to the PVM to determine its baseline CPU feature set. Concurrently, the bytecode tracking module analyzes the application's bytecode to produce a call graph and identify all symbolic calls to native libraries. The outputs of these two tracks are then bridged by the symbol resolution module. It maps the symbolic calls from the bytecode to their implementations within the respective native libraries. Each successfully resolved native function is subsequently traced by the native library tracking module to determine its own feature requirements. Finally, the feature sets from the PVM and all invoked native libraries are consolidated into a single set that represents the total hardware requirements of the workload.

C. Compatibility Checker

The final stage in the Pygration pipeline is performed by the *Compatibility Checker*. This module synthesizes the data from the preceding components to make the migration decision. Its core function is a set-theoretic comparison that verifies if the workload-specific feature set, provided by the *Workload Instruction Analyzer*, is a subset of the features available on a potential destination instance, as determined by the *CPU Feature Collector*. This workload-aware strategy stands in contrast to the conservative approach of CRIU, which compares the full feature set of the source instance to the candidate destination instances. By focusing only on the features that the workload actually requires, the proposed method significantly reduces the probability of false-negative detections, thereby maximizing the pool of viable migration targets without compromising reliability.

V. Issues During Development

When analyzing the bytecode and native libraries to extract CPU features used by a process, multiple issues arose, and we summarize the issues and how we addressed them.

A. Handling Inconsistencies in TSX Deprecation

A subtle challenge for compatibility checking arises from the inconsistent hardware deprecation of Intel's Transactional Synchronization Extensions (TSX) [26], [27]. TSX, identified by the `RTM` and `HLE` CPUID flags, was disabled on many CPUs via microcode updates in response to security vulnerabilities [28], [29]. When disabled, the `RTM` and `HLE` flags are cleared, and core transactional instructions like `xbegin` and `xend` are rendered non-operational. However, this deprecation process can be inconsistent. The `xtest` instruction, which checks if the processor is in a transactional state, is not always disabled along with the rest of the feature set. While a transactional region cannot be entered, the `xtest` instruction itself may still execute without an issue, always indicating that the processor is not in a transactional state. This creates a difficult corner case: a workload that only utilizes the `xtest` instruction would run correctly on a host with disabled TSX, but a checker that relies solely on the `RTM` and `HLE` flags would incorrectly deem it incompatible. To address this potential for false negatives, the proposed system extends its feature detection mechanism. The CPU Feature Collector not only queries the standard `RTM` and `HLE` flags but also performs a separate, direct check to determine if the `xtest` instruction can be executed without causing a fault. The result is stored as an independent feature flag. Consequently, the Compatibility Checker can distinguish between workloads that require the full TSX functionality versus those that only require the ability to execute the `xtest` instruction, thereby avoiding erroneous incompatibility detections.

B. Impact of Dynamic Linker Binding Strategies

The timing of symbol resolution for dynamically linked libraries directly impacts the completeness of the static analysis. By default, many systems employ lazy binding as a heuristic for performance optimization, where the address of an external library function is not resolved until its first invocation. In this mode, the GOT entry for an unresolved function points to a resolver routine within the dynamic linker. This deferred resolution reduces application startup time. This behavior, however, poses a challenge for Execution Path Tracking. As a static analyzer, it inspects the process's memory state at a

single point in time. If a function has not yet been called, its true address will not be present in the GOT, rendering that execution path invisible and leading to an incomplete call graph. To circumvent this limitation, the analysis requires that applications use immediate binding, which is also known as now binding. This strategy, often enabled by setting the `LD_BIND_NOW=1` environment variable, forces the dynamic linker to resolve all symbol addresses at program launch. This ensures that all potential execution paths into shared libraries are visible to the static analysis from the outset.

C. Limitations of Native Library Tracking

While the proposed Execution Path Tracking heuristic for native library analysis is more precise than a full-text segment scan, as a static analysis technique, it has inherent limitations in handling execution paths that are determined dynamically at runtime.

a) Indirect Function Calls: The tracker cannot resolve branch targets that are not specified as immediate values in the instruction itself. This is most common in function calls made via pointers or registers, such as the `call eax` instruction. The value of a general-purpose register like `EAX` is determined by the program's dynamic state and can change continuously. Accurately determining the target of such a call would require runtime instrumentation, which incurs significant overhead and is outside the scope of this static approach. Consequently, execution paths initiated through function pointers or C++ virtual method tables cannot be reliably traced.

b) Dynamically Loaded Libraries: The analysis is limited to the libraries that are loaded into the process's memory space at the time of inspection. Modern applications often use lazy loading, via mechanisms like the `dlopen()` system call [30], [31], to load shared libraries on demand. While this approach optimizes application startup time and memory usage, it poses a challenge for static analysis. If a library is loaded after analysis is performed, its code and any unique CPU feature dependencies within it might be missed.

D. Limitations of Bytecode Tracking

As a static analysis technique, the proposed bytecode tracking module has inherent limitations when faced with Python code that determines dependencies dynamically at runtime.

a) Dynamic Module Importing: A primary challenge arises from dynamic imports, such as those using the `importlib.import_module()` function or loading native libraries with `ctypes.CDLL()`. In these cases, the name or path of the module to be loaded is often stored in a variable. If the value of this variable is not a constant literal but is instead constructed at runtime (e.g., from a configuration file, user input, or string manipulation), a static analyzer cannot determine which library will be loaded.

b) Namespace Ambiguity from Wildcard Imports: While wildcard imports (e.g., `from module import *`) can be tracked, they can introduce namespace ambiguities that are difficult to resolve statically. If two different modules, both imported via a wildcard, define a function with the same name, a subsequent call to the function becomes ambiguous. Determining the correct target depends on the import order and potential symbol overwriting, which is complex to model without execution, and it can lead to incomplete call tracking.

E. Applicability to Different Python Workload Types

Given the static analysis limitations, it is important to characterize the types of Python workloads for which Pygration is expected to perform well and those where its coverage may be reduced. Pygration is well-suited for the majority of compute-intensive Python applications, particularly those in scientific computing, machine learning, and data processing. These workloads typically invoke native libraries through well-defined FFI mechanisms such as `ctypes`, Cython, or the Python C API, all of which Pygration's symbol resolution can handle. The 14 workloads evaluated in this study, spanning machine learning frameworks (`sklearn`, `XGBoost`, `llm`), web frameworks (`FastAPI`, `Falcon`), and data processing tools (`Dask`, `Matplotlib`), are representative of this category and were all analyzed successfully.

Conversely, workloads that rely heavily on runtime-determined dependencies may experience reduced analysis coverage. This includes applications with plugin architectures that dynamically load native libraries via `dlopen()` based on runtime configuration, or frameworks that heavily use metaprogramming to construct and dispatch native function calls dynamically. Prior work has shown that Python's dynamic features such as runtime attribute lookup and metaprogramming fundamentally limit static call graph construction, achieving only approximately 70% recall [32], and the Python-to-native-C interface remains particularly challenging for single-language static analyzers [33]. In such cases, the static analysis may not capture all native library dependencies. However, even for these workloads, Pygration's analysis of the PVM and statically resolvable libraries still provides a substantial improvement over the conservative CRIU baseline. Furthermore, when stronger safety guarantees are required, users can configure Pygration to use the Full Scan heuristic as a fallback, which trades recall for more complete coverage.

VI. Evaluation

This section presents a comprehensive evaluation of the proposed migration compatibility detector, Pygration. The experiments are designed to assess the system's accuracy in identifying compatible instances, its performance overhead, and its practical benefits in real-world cloud scenarios. Specifically, the evaluation aims to answer the following research questions:

1) **RQ1 (Accuracy):** To what extent does the proposed system improve the accuracy of migratable instance detection, measured in precision and recall, relative to the baseline host-level checking approach adopted by CRIU?
2) **RQ2 (Performance Overhead):** What is the computational overhead of the proposed analysis pipeline, and how does it affect the system's feasibility for real-time migratable instance detection?

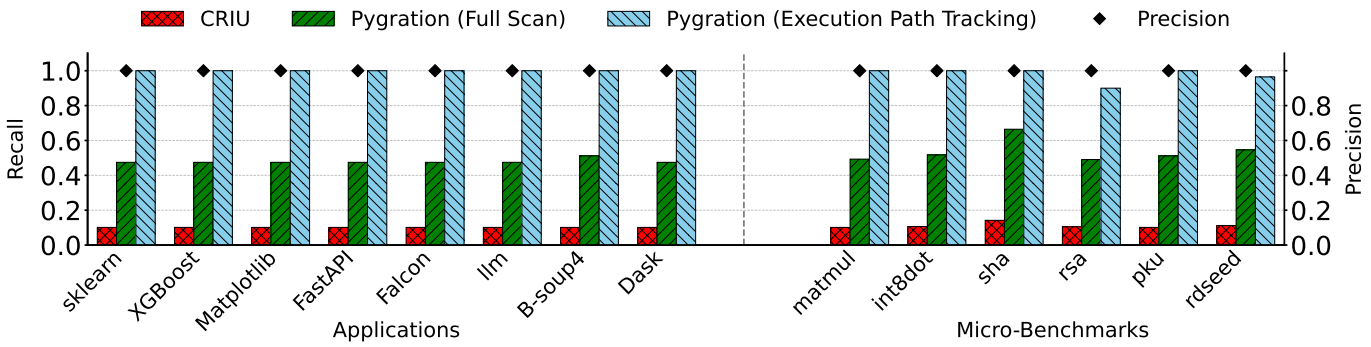


Fig. 8. Comparison of CRIU and the proposed method, Pygration with two different native library scan heuristics, Full Scan and Execution Path Tracking.

3) **RQ3 (Real-World Efficacy):** What is the quantifiable impact of the system's improved detection recall on key operational metrics, such as cost savings and service reliability, especially when it is applied to a cloud spot instance environment?

A. Environment Setup

The experimental testbed was constructed based on the CPU feature maps of 522 unique AWS EC2 x86-64 instance types. These feature maps were gathered using the *CPU Feature Collector* module detailed in Section IV-A. Exhaustively verifying directional migration compatibility between all possible pairs of these n instances would require $n \times (n-1)$ experiments, which for 522 instance types amounts to over 270,000 migration tests per workload. Such a scale is prohibitive in terms of both time and cost. To create a tractable experimental design, the 522 instance types were first clustered into 19 distinct groups based on their identical CPU feature sets. This grouping enabled a two-tiered validation approach. First, intra-group compatibility was confirmed by performing all-to-all migration tests within sample groups. For the main experiment, inter-group compatibility was evaluated by selecting one representative, cost-effective instance from each of the 19 groups. This resulted in a comprehensive set of $19 \times 18 = 342$ unique, directional migration paths to be tested for each workload. The evaluation was performed across 14 diverse and commonly used Python applications that are broadly divided into Applications and Micro-Benchmarks. The Applications category includes machine learning libraries (`sklearn` [34], `XGBoost` [35], and `llama-cpp-python` [36] hereafter `llm`), web frameworks (`FastAPI`, and `Falcon`), and data processing and visualization tools (`Dask`, `beautifulsoup4`, and `Matplotlib`). The machine learning workloads include `sklearn`, which performs classification with Random Forest [37], Support Vector Machine, and Logistic Regression, regression with Random Forest and Linear Regression, and clustering with K-Means; `XGBoost`, which trains a gradient boosting model on the MNIST dataset; and `llm`, which executes inference using a small-scale language model [38]. The web frameworks, `FastAPI` and `Falcon`, run a simple server to process HTTP requests. For data processing and visualization, `Dask` evaluates dynamic task graphs [39], `beautifulsoup4` parses a static HTML document, and `Matplotlib` generates line plots of sine and cosine waves and histograms of random data distributions. In contrast to the Applications, Micro-Benchmarks workloads focus on evaluating specific CPU features. `matmul` performs a 2x3 by 3x2 matrix multiplication using the OpenBLAS library [40] imported from the numpy library. `int8dot` calculates the dot product of 8-bit integer vectors, utilizing the most advanced SIMD instruction set available on the CPU, such as *AVX512-VNNI* or *AVX2*. `sha` and `rsa` evaluate cryptographic performance by performing SHA-256 hashing and RSA encryption, respectively, which can leverage hardware acceleration. `pku` tests the Memory Protection Keys (PKU) feature by manipulating memory access permissions. Lastly, `rdseed` utilizes the CPU's hardware random number generator by directly calling the *RDSEED* instruction.

B. RQ1 : Migratable Instance Detection Accuracy

The first experiment addresses RQ1 by quantitatively evaluating the accuracy of the proposed compatibility detection methods. The evaluation is based on two standard metrics of precision and recall, which are defined in the context of this work as follows:

- **Precision** measures the safety of the detector. It is the fraction of predicted compatible instances that are truly compatible. A perfect precision of 1.0 is critical, as any value lower than this indicates a false-positive prediction that would lead to a migration failure.
- **Recall** measures the completeness of the detector. It is the fraction of all truly compatible instances in the testbed that were successfully identified. A low recall signifies an overly conservative approach that unnecessarily restricts the pool of viable migration targets, limiting flexibility and potential benefits.

1) Precision and Recall of Pygration: Figure 8 presents a comparison of precision and recall between the proposed Pygration system and the baseline CRIU approach. The CRIU method performs a conservative, host-level check based on the CPU features advertised by the host machine. For this evaluation, two distinct heuristics for Pygration's native library analysis were assessed: *Full Scan* and *Execution Path Tracking*.

a) Precision: As shown by the circular markers in the figure whose values are shown in the secondary vertical axis, all three evaluated methods, *CRIU*, *Pygration (Full Scan)*, and *Pygration (Execution Path Tracking)*, achieved a perfect precision score of 1.0 across all workloads. This critical result confirms that none of the methods produce false positives; if an instance is identified as compatible, the migration is guaranteed to succeed without a crash. This establishes that both of the proposed Pygration heuristics maintain the same level of safety as the conservative CRIU baseline.

b) Recall: In contrast, the recall results, shown by the bars whose values are shown in the primary vertical axis, reveal significant differences for each method. The baseline *CRIU* approach exhibited a consistently poor recall of approximately 0.10 across all workloads, failing to identify nearly 90% of all viable migration targets. The *Pygration (Full Scan)* heuristic provided a substantial improvement, with recall values typically ranging from 0.4 to 0.7. While this demonstrates the benefit of being workload-aware, analyzing all code within the process's text segments still includes dependencies from unused functions, leading to the unnecessary exclusion of many compatible instances. The *Pygration (Execution Path Tracking)* heuristic achieved superior performance, consistently achieving a perfect recall for all tested workloads except RSA and RDSEED workloads. A thorough analysis revealed a discrepancy between the CPU features advertised by the hypervisor (via CPUID) and the actual hardware capabilities. It is important to distinguish between these two levels of capability: the *hardware-level capability*, which refers to the physical CPU's supported features, and the *provider-exposed capability*, which refers to the features reported by the hypervisor's virtualized CPUID to the guest VM. For some instance types, the *ADOX_ADCX* feature for RSA workload and *RDSEED* feature for RDSEED workload were reported as unavailable at the provider-exposed level, yet the underlying hardware was, in fact, capable of executing them. A hypervisor can manipulate a list of CPU features to be exposed to guest virtual machines, which is called CPU feature masking [41]. Cloud vendors may use this feature to mask some CPU features to ensure compatibility and enable live migration across different hardware generations [42]. Pygration operates on provider-exposed capabilities, which is the correct conservative choice since these are the only features guaranteed to be available to the guest VM. This design leads to a slight degradation in recall in the affected cases. More importantly, this behavior does not impact precision, ensuring that no migration target identified as compatible will result in a failure. If cloud providers were to offer a more transparent CPUID reporting mechanism, Pygration's recall in these edge cases would improve further.

2) Ablation Study of Pygration Components: To quantify the individual contribution of each analysis component, an ablation study was conducted. The complete **Pygration** system (`PVM + Bytecode Tracking`) was compared against two ablated versions: a `PVM Tracking Only` configuration, which omits the analysis of invoked native libraries from application Python code, and a `Bytecode Tracking Only` configuration, which omits the baseline analysis of the PVM itself. Table I summarizes the results for workloads where the ablated configurations failed to achieve perfect precision or recall, along with the root cause of the failure.

TABLE I
ABLATION STUDY OF THE PROPOSED SYSTEM FOR THE IMPACT OF PRECISION, RECALL, AND ERROR CAUSES

Method	Workload	Precision	Recall	Cause
PVM Tracking Only	int8dot	0.9539	1.0000	AVX512_VNNI
	rdseed	0.9032	1.0000	RDSEED
	rsa	0.9677	1.0000	ADOX_ADCX
	sha	0.7143	1.0000	SHA
Bytecode Tracking Only	B-soup4	0.7092	1.0000	No native library calls to track
	Falcon	0.7092	1.0000	
	FastAPI	0.7092	1.0000	
	rdseed	1.0000	0.9643	RDSEED
	rsa	1.0000	0.9000	ADOX_ADCX

a) Impact on Precision (Safety): The results from the *Precision* column demonstrate that both the PVM and bytecode tracking components are critical for ensuring migration safety. The complete, integrated Pygration system was the only configuration to achieve a perfect precision score of 1.0 across all workloads. Both ablated configurations produced false positives under different conditions. The `PVM Tracking Only` configuration failed for workloads with specialized native libraries, such as `int8dot`, where its precision dropped to 0.9539 because it was unaware of the library's `AVX512_VNNI` dependency. Conversely, the `Bytecode Tracking Only` configuration failed for applications like `Beautifulsoup4`, `FastAPI`, and `Falcon` with a precision of 0.7092. As noted in the table, these workloads do not make explicit native library calls from the Python code. Consequently, the analyzer fails to account for the baseline CPU requirements of the PVM itself and its underlying dependencies, such as `libc`, leading to an unsafe migration decision if the target host is incompatible with the interpreter.

b) Impact on Recall (Completeness): The results in the *Recall* column highlight a notable anomaly caused by hypervisor feature masking, where a more thorough analysis method results in a lower recall score, as discussed in Section VI-B1. The `PVM Tracking Only` configuration, which is oblivious to library-specific dependencies, did not detect the masked `ADOX_ADCX` and `RDSEED` features. It consequently produced a less restrictive requirement set, leading to a perfect recall of 1.0. In contrast, both the `Bytecode Tracking Only` configuration and the complete Pygration system correctly identified these dependencies. By strictly adhering to the features enumerated by the `CPUID` instruction, these methods flagged migrations to the instances with the masked features as incompatible. Although this is the correct behavior for a safety-critical system, these predictions are categorized as false negatives in the ground truth evaluation, resulting in the observed recall degradation (e.g., to 0.9000 for `rsa`). This outcome demonstrates a deliberate design trade-off: Pygration prioritizes precision (safety), accepting a minor reduction in recall in corner cases involving discrepancies between advertised hardware capabilities.

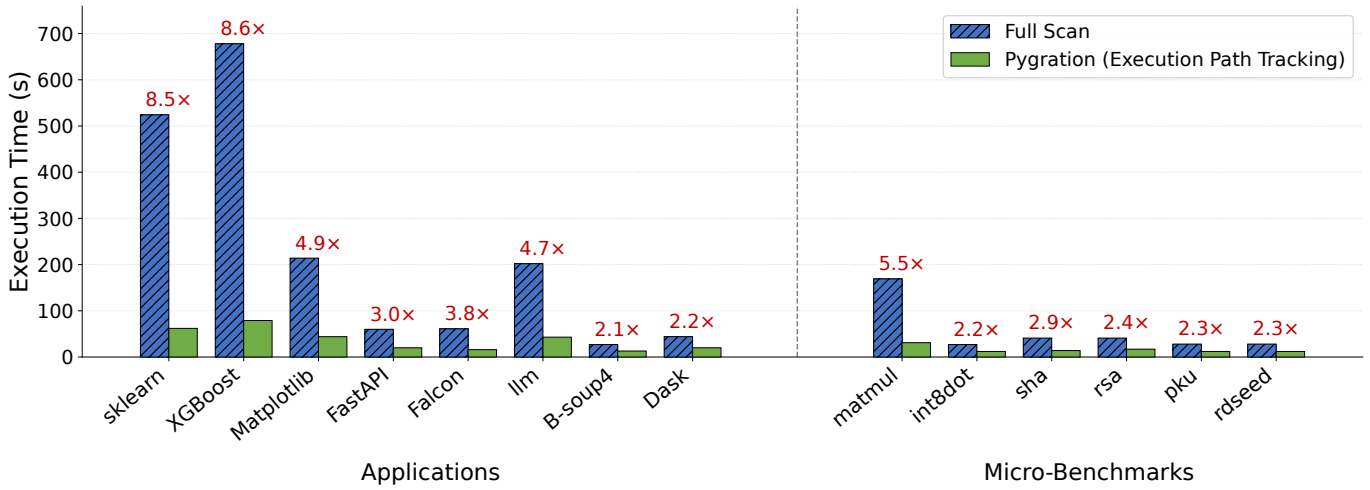


Fig. 9. The execution time to detect migratable instances when using Full Scan method and Pygration with Execution Path Tracking (EPT). The y-axis shows execution time in seconds, and the speedup ratio of EPT over the Full Scan is annotated above each pair of bars.

C. RQ2: Proposed System Operation Overhead

This section addresses RQ2 by evaluating the computational overhead of the proposed system. Figure 9 compares the analysis latency of the *Pygration (Execution Path Tracking - EPT)* heuristic against the *Full Scan* method. The y-axis shows the execution time in seconds for each method. Full Scan and Pygration (EPT) are presented as grouped bars for direct comparison, with the speedup ratio annotated above each pair. All measurements were performed on an AWS `c5.xlarge` instance.

The results clearly show that the EPT heuristic provides a significant speedup, ranging from approximately $2.1\times$ for simple applications to over $8.6\times$ for complex ones. The performance improvement is most pronounced for workloads with large native libraries, such as `XGBoost` ($8.6\times$ speedup) and `sklearn` ($8.5\times$ speedup). This is because the Full Scan method must wastefully analyze the entire multi-megabyte text segment of libraries, whereas EPT intelligently traces only the small fraction of functions actually invoked by the workload.

Finally, the latency for Pygration using the EPT heuristic ranged from 11.9 seconds (`rdseed`) to 79.2 seconds (`XGBoost`), depending on workload complexity. While a latency of tens of seconds may be too high for immediate migration upon a preemption warning of spot instances, Pygration is designed for *proactive, offline profiling* rather than on-the-critical-path analysis. In a typical deployment, Pygration runs as a background process that periodically generates and updates a pre-computed list of compatible migration targets. When a spot instance interruption warning is received (typically 2 minutes in advance on AWS), the system can instantly consult this pre-computed list to select an optimal target, thereby completely decoupling the analysis latency from the migration response time.

As a direction for future work, combining Pygration's static analysis with lightweight runtime instrumentation techniques (e.g., recording actually-invoked native functions via `LD_AUDIT`) could further refine the required CPU feature set with minimal overhead and improve coverage for dynamically resolved function calls.

Figure 10 presents the memory usage incurred during the analysis of various Python workloads. The vertical axis shows memory consumption in megabytes (MB), broken down by component in a stacked bar format. *GDB* represents the peak memory usage after the target workload has been loaded for analysis. *PVM Tracking* represents additional memory usage that is consumed from the tracking of the entry point of a Python interpreter binary, *_start* function. *Bytecode Tracking* represents the additional memory consumed during tracking bytecode execution and native libraries invoked from the bytecode. Both PVM and Bytecode tracking use Execution Path Tracking for native library analysis.

The results show that the baseline GDB overhead is the largest component, as it includes the memory footprint of the target application. This component's size is proportional to the workload's complexity, ranging from approximately 150-200 MB for the micro-benchmarks to over 500 MB for a large application like `XGBoost`.

After loading an application using GDB, it consumes an additional 130MB when tracking PVM library, which does not show a different pattern across distinct workloads. This overhead consists of a fixed component for analyzing the PVM itself, in which 1663 functions were tracked, and a variable component for analyzing the invoked native libraries from PVM.

In contrast, the Bytecode Tracking component's overhead is highly variable and depends on the application's scale. For instance, the analysis of `xgboost`, which involves parsing 1224 Python modules, required approximately 150 MB for this stage. This is a 17-fold increase compared to the `matmul` benchmark, which only required the analysis of 111 modules. This demonstrates that the memory overhead of this component scales with the number of Python modules to be parsed and native functions to be traced.

Figure 11 illustrates the number of unique CPU features that each detection method considers when determining com-

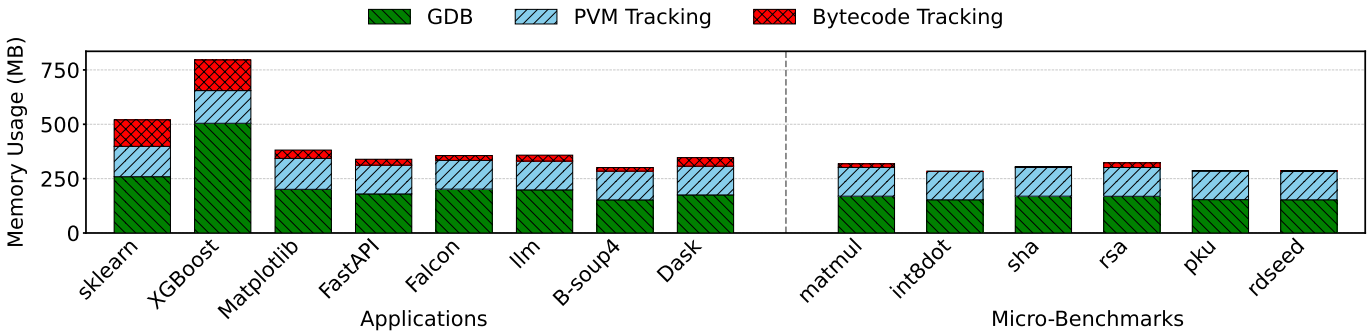


Fig. 10. Memory consumption while tracking various workloads using Pygration with Execution Path Tracking module

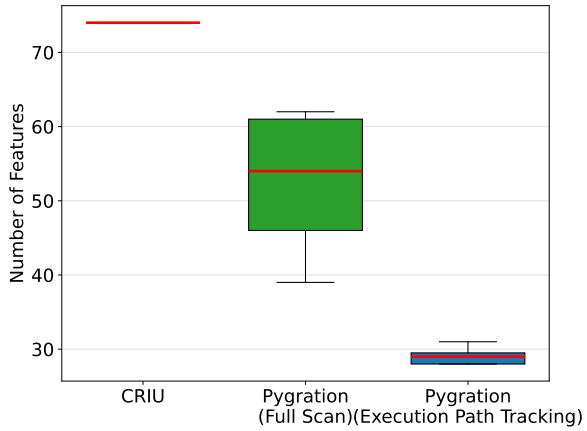


Fig. 11. Comparing the number of CPU features considered during migratable instance detection. Proposed heuristics lower the number of CPU features that are considered during migratable instance detection, resulting in an improvement in the recall without compromising the precision

Workload	Traceable	Untraceable	Ratio
B-soup4	11,111	712	0.060
Dask	11,112	712	0.060
Falcon	11,111	712	0.060
FastAPI	11,111	712	0.060
int8dot	11,145	712	0.060
llm	11,155	714	0.060
matmul	14,378	924	0.060
Matplotlib	13,751	863	0.059
pku	11,157	712	0.060
rdseed	11,116	712	0.060
rsa	13,347	822	0.058
sha	11,229	722	0.060
sklearn	15,459	1,019	0.062
xgboost	14,182	910	0.060

TABLE II
THE NUMBER OF TRACEABLE AND NON-TRACEABLE FUNCTION CALL INSTRUCTIONS WHEN USING THE EXECUTION PATH TRACKING MODULE

patibility. The baseline CRIU method is workload-agnostic, consistently checking a fixed set of 74 features for every application. In contrast, the proposed workload-aware heuristics significantly reduce this number. The Full Scan method narrows the required feature set to a range of 39 to 62, depending on the application. The more precise EPT heuristic is even more effective, further reducing the set to just 28 to 31 features. This significant reduction in the number of feature constraints is the primary reason for the substantial improvement in recall demonstrated by the proposed system, as it eliminates irrelevant requirements and thereby expands the pool of compatible migration targets.

Table II presents the number of traceable and untraceable function call instructions and the ratio of untraceable to traceable when using EPT. Unlike the text-segment full-scan method, execution path tracking may lead to tracking omissions in certain situations, such as calling a function by a register value. It can happen when calling a function via pointers or using Virtual Tables to override class virtual functions in C++. Such cases impose a potential risk of tracking misses, which can lead to a process crash after migration. Although we were unable to observe any crashes during the experiments (the migration success detection precision is 1.0), we demonstrate how many functions were untraceable for various workloads. Among all functions, on average, less than 6% of the functions were untraceable due to the reference to the register value in a function call.

Although there was no false-positive detection, to make the function tracking complete, further research is necessary to address the untraceable function problem. To mitigate this risk in practice, Pygration provides a configurable safety/recall trade-off. If tracking misses are not tolerable for a given deployment, one can apply the text-segment full scan heuristic, whose recall is lower than the execution path tracking but still significantly better than the default CRIU implementation. This design allows users to select the analysis mode that best matches their reliability requirements. Additionally, as a practical deployment strategy, a lightweight post-migration liveness probe can be employed to verify correct execution immediately after restoration. If such a probe detects a failure (e.g., via SIGILL), the system can trigger a re-migration to a known-safe instance from the pre-computed compatible target list, thereby providing a recovery mechanism that further enhances the robustness of the migration process.

To evaluate the efficiency of the proposed bytecode tracking module, a comparative performance analysis was conducted against PyCG [32], which is a Python source code static analysis tool that generates call graphs by parsing the Abstract Syntax Tree (AST) [43]. The performance of both tools was measured across our benchmark suite of Python workloads, focusing on two key metrics of total execution time and peak memory consumption.

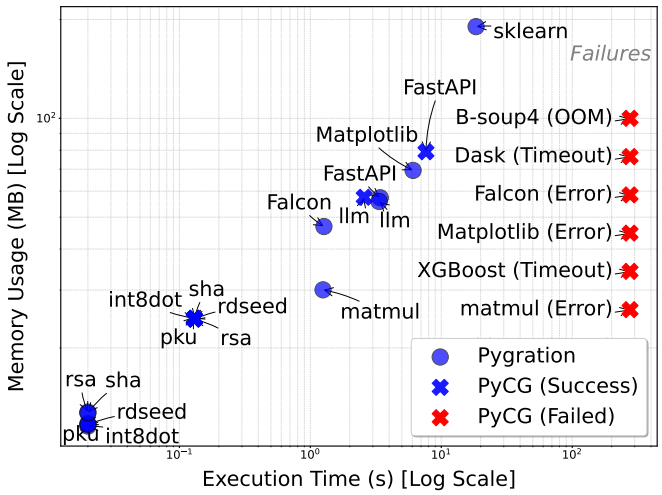
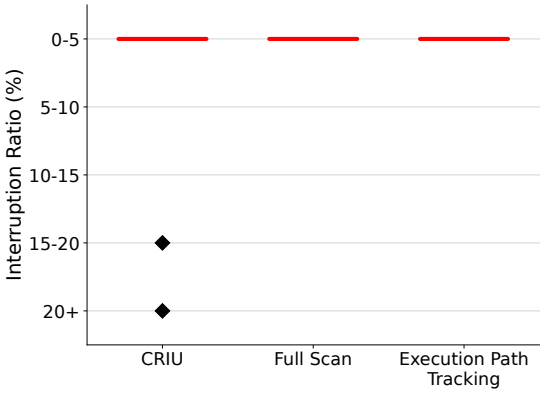


Fig. 12. Comparing the Python source code analysis module of the proposed Bytecode tracking with an AST-based tool, PyCG.

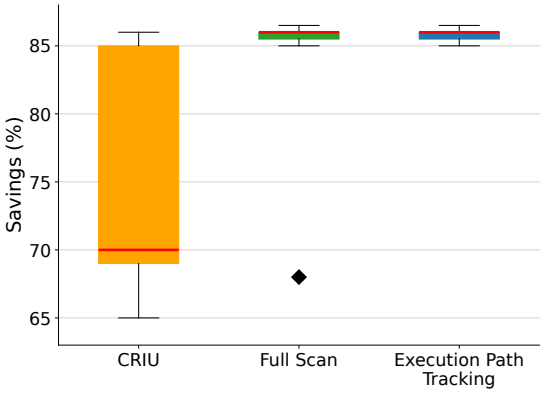
The results are presented in Figure 12 using a scatter plot format. In the figure, Pygration is marked with circle markers, and PyCG is marked with X markers. For some workloads, PyCG could not complete within 2.5 hours or resulted in an out-of-memory problem, and we mark such cases using red colors. The results demonstrate that the proposed bytecode-based approach is significantly more efficient and robust than the AST-based method. For many complex workloads with extensive dependencies, such as `sklearn` and `matplotlib`, PyCG failed to complete the analysis, encountering runtime errors or memory exhaustion. For other large workloads like `xgboost`, PyCG timed out after 2.5 hours. In contrast, our proposed bytecode tracking algorithm successfully analyzed all workloads, completing the analysis for `xgboost` in just 17.88 seconds while consuming approximately 210 MB of memory.

This substantial performance disparity stems from the fundamental differences in methodology. AST-based tools like PyCG must first perform a computationally expensive parsing step to construct a complex tree representation of the entire codebase. Resolving call dependencies often requires multiple, recursive traversals of this tree, leading to high computational and memory overhead. Our bytecode-based approach bypasses this expensive parsing and tree-construction phase. It operates directly on the simpler, linear instruction sequence already generated by the Python interpreter, enabling a more efficient, single-pass analysis to construct the call graph and resulting in substantially lower overhead.



(a) Percentage of interruptions (%) (lower value, higher in the vertical axis, is better)

(b) Spot instance availability, T3 score (higher is better)



(c) Cost savings (%) (higher is better)

Fig. 13. Enhancements in the reliability and cost savings when the proposed system is applied in a spot instance environment

D. RQ3: Efficacy in a Spot Instance Environment

To answer RQ3, a simulation was conducted using historical data from the SpotLake dataset [19] to quantify the potential benefits of the proposed system in a cloud spot instance environment, focusing on reliability and cost savings. A larger pool of compatible migration targets allows a user to be more selective, choosing an instance with the most favorable characteristics. For example, a matrix multiplication workload using the OpenBLAS library in an AWS *c6a.24xlarge* instance type can be migrated to eight distinct instance types when using the CRIU implementation. When using the proposed heuristic, the number of migratable instances increases to 520, which can provide significant flexibility when choosing target instances.

To quantitatively evaluate the enhanced performance efficiency, compatibility checks were performed for all workloads by having 19 groups with unique CPU features as migration source instances. The target instances were selected from

among 522 instance types of the same size or larger. For each source, the set of compatible targets was determined using CRIU, Full Scan, and Execution Path Tracking. From the resulting set for each method, an optimal target was selected based on three metrics derived from the SpotLake dataset [19]: interruption ratio (lower is better), availability (higher is better), and cost savings (higher is better). The results are presented in Figure 13. It is important to note that these results are based on historical spot instance data rather than end-to-end experiments with real spot interruptions; the actual benefits may vary depending on real-time market conditions.

a) Enhanced Reliability: As shown in Figure 13a, the proposed methods indicate a substantial improvement in reliability. AWS provides the interruption ratio of spot instances, and we use the dataset to infer the reliability of spot instances. The interruption rate refers to the frequency at which a particular spot instance was forcibly terminated by the provider before a user-initiated shutdown within the past month. The advertised interruption ratio is categorized as less than 5%, $5\%-10\%$, $10\%-15\%$, $15\%-20\%$, and more than 20%. Both Full Scan and Execution Path Tracking consistently identified compatible instances in the lowest interruption ratio category of less than 5%. In contrast, the restricted candidate pool from CRIU often forced the selection of less reliable instances; in 9% of the test cases, CRIU selected spot instances whose interruption ratio is 15%-20%, and 4.5% with a rate exceeding 20%

b) Increased Availability: Figure 13b presents the spot instance availability metric, Spot Placement Score (SPS) [44], [45], when using the proposed heuristics. Unlike the interrupt ratio from the prior month, the SPS dataset reflects the instantaneous availability information. The score is an integer value ranging from 1 to 3, where the higher value indicates higher availability. The score is offered for multiple spot instance requests, and SpotLake [46] provides the *T3* metric, which represents the maximum number of target instances whose SPS score remains three. The higher T3 value implies that a larger number of spot instances can be offered reliably. The larger candidate pools from the proposed methods enabled the selection of instances with significantly higher and more consistent availability scores compared to CRIU.

c) Maximized Cost Savings: The most direct financial benefit is shown in the cost savings analysis in Figure 13c. The median cost savings for Pygration was 86% relative to on-demand prices, a substantial improvement over CRIU's 70%. The advantage was even more pronounced in the worst-case scenario, where Pygration maintained 85% savings, while CRIU's savings dropped to 65%. In conclusion, by significantly expanding the pool of compatible instances, Pygration allows users to consistently choose spot instances that are more reliable, more readily available, and offer greater cost savings.

Integrating Pygration with a full migration controller that automates the end-to-end workflow of interruption detection, checkpointing, target selection from Pygration's pre-computed list, and restoration is a natural next step for future deployment.

VII. Related Work

Barbalace et al. [47] introduced compiler and operating system extensions that enable the migration of the execution runtime between heterogeneous ISA servers comprising X86 and ARM. The authors presented a new multi-ISA binary architecture for the efficient migration of natively compiled applications. The work was further demonstrated in edge-computing environments [48]. Although the approaches allowed migration between various ISA machines, they do not support the migration of special features, such as SIMD extensions and setjmp/longjmp. To the best of the authors' knowledge, automatic conversion of execution runtimes with special instructions is not yet possible, and this work is the first attempt to detect migratable instances between machines with distinct CPU features. Few studies have applied the live migration process to enhance computing resource usage efficiency. Juric et al. [49] demonstrated the full implementation of user-specific checkpointing, restoration, and real-time migration functionalities for JupyterHub in a cloud environment. The authors presented a solution for cloud deployment that improves the user experience and reduces operational costs through container migration [2], [3]. Their objective is to minimize resource waste when user sessions are inactive. Cunha et al. [50] pointed out that not all cells in the Jupyter Notebook need to run in the same environment. Although some cells require high computational power, others may not, and it is crucial to categorize cells and migrate them to the appropriate environments. For the aforementioned works, the proposed migratable instance detection algorithm can increase the number of target instances, significantly boosting migration efficiency. Salis et al. [32] proposed PyCG, a static analysis tool for generating the call graph of Python programs. PyCG effectively handles Python's dynamic features, modules, and higher-order functions using AST-based analysis and outperforms existing tools such as Pyan [51] and Depends [52]. Additionally, PyCG demonstrated its applicability in dependency impact analysis and security vulnerability tracking. However, PyCG includes functions in the call graph that are not part of the execution path and fails to track call paths for external modules when source code is unavailable. Furthermore, it exhibits significant operational overhead and is difficult to apply to a large-scale library, such as XGBoost [35].

VIII. Conclusion

This paper addressed the critical challenge of conservative CPU compatibility checking, which severely limits the efficiency and flexibility of live migration for Python applications in the cloud. The host-level approach, implemented by the process migration tool CRIU, is workload-agnostic and unnecessarily disqualifies a vast number of viable migration targets. This problem is particularly severe for Python, where the multi-layered execution model involving the PVM, bytecode, and native libraries obscures the true hardware dependencies of a workload. To solve this problem, we introduced Pygration, a novel, workload-aware system that performs a deep, cross-layer analysis. By integrating Python bytecode tracking with a precise native library tracking heuristic, Pygration accurately

identifies the minimal set of CPU features an application actually requires. This allows it to dramatically expand the pool of compatible migration instances while guaranteeing safety. Our comprehensive evaluation on 522 unique AWS instance types demonstrated the effectiveness of this approach. Pygration achieved a perfect precision of 1.0, ensuring no migration failures, while improving recall by over 5× compared to the CRIU baseline. Furthermore, a simulation in a spot instance environment showed that this improved accuracy translates into tangible benefits, enabling a 16% improvement in median cost savings by allowing users to select more reliable and cost-effective instances. These results confirm that Pygration provides a robust and practical solution, enabling more efficient, flexible, and cost-effective live migration for Python workloads in heterogeneous cloud environments.

Acknowledgment

This work is supported by the National Research Foundation (NRF) Grant funded by the Korean Government (NRF-2020R1A2C1102544), by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (RS-2022-00144309 & RS-2025-25441560), by the research fund of Hanyang University (HY-202500000001053), and AWS Cloud Credits for Research program.

References

[1] Chandra Prakash, Debadatta Mishra, Purushottam Kulkarni, and Umesh Bellur. Portkey: Hypervisor-assisted container migration in nested cloud environments. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2022, page 3–17, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete container state migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142, 2017.

[3] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

[5] Guikun Wang, Bin Wen, Jingtao He, and Qingbin Meng. A new approach to reduce energy consumption in priority live migration of services based on green cloud computing. *Cluster Computing*, 28(3), Jan 2025.

[6] Soramichi Akiyama, Takahiro Hirofuchi, and Shinichi Honiden. Evaluating impact of live migration on data center energy saving. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 759–762, 2014.

[7] Tae Seung Kang, Maurício Tsugawa, Andréa Matsunaga, Takahiro Hirofuchi, and José A.B. Fortes. Design and implementation of middleware for cloud disaster recovery via virtual machine migration management. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 166–175, 2014.

[8] Fei Zhang, Guangming Liu, Xiaoming Fu, and Ramin Yahyapour. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Communications Surveys Tutorials*, 20(2):1206–1243, 2018.

[9] Adrian Reber. Container migration with podman on rhel. https://www.redhat.com/en/blog/container-migration-podman-rhel, 2019.

[10] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, dec 2003.

[11] Chris Currier. *Protocol Buffers*, pages 223–260. Springer International Publishing, Cham, 2022.

[12] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 329–341, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 71–84, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Prateek Sharma, David Irwin, and Prashant Shenoy. How not to bid the cloud. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[15] K. Lee and M. Son. Deepspotcloud: Leveraging cross-region gpu spot instances for deep learning. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 98–105, 2017.

[16] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Terry Yang, Victor Rühle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Snape: Reliable and low-cost computing with mixture of spot and on-demand vms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 631–643, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Alexandra Vintila, Ana-Maria Oprescu, and Thilo Kielmann. Fast (re-)configuration of mixed on-demand and spot instance pools for high-throughput computing. In *Proceedings of the First ACM Workshop on Optimization Techniques for Resources Management in Clouds*, ORMaCloud '13, page 25–32, New York, NY, USA, 2013. Association for Computing Machinery.

[18] Junho Lim, KyungHwan Kim, and Kyungyong Lee. Workload-aware live migratable cloud instance detector. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 178–188, 2024.

[19] S. Lee, J. Hwang, and K. Lee. Spotlake: Diverse spot instance dataset archive service. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 242–255, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[20] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[21] intelxed. Intel encoder-decoder (xed). https://github.com/intelxed/xed, 2025.

[22] Capstone, the ultimate disassembler. https://www.capstone-engine.org/, 2025.

[23] Ludovic Courtès. Reproducibility and performance: Why choose? *Computing in Science Engineering*, 24(3):77–80, 2022.

[24] Python Documentation. ctypes — a foreign function library for python, 2025.

[25] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[26] Zixian Cai, Stephen M. Blackburn, and Michael D. Bond. Understanding and utilizing hardware transactional memory capacity. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021, page 1–14, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2013.

[28] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 380–392, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision l3 cache attack using intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, Vancouver, BC, August 2017. USENIX Association.

[30] Michael Kircher. Lazy acquisition. In *EuroPLoP*, pages 151–164, 2001.

[31] RuiHeng Tang, Fei Liu, and Xu Xiao. A lightweight approach for large cad models based on lazy loading. In *2023 IEEE 18th Conference on Industrial Electronics and Applications (ICIEA)*, pages 1977–1982, 2023.

[32] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1646–1657, 2021.

[33] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multi-language static analysis of python programs with native c extensions. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 323–345, Cham, 2021. Springer International Publishing.

[34] 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*, 12:2825–2830, 2011.

[35] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Python bindings for llama.cpp. https://llama-cpp-python.readthedocs.io/en/latest/, 2025.

[37] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[39] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, number 130-136. Citeseer, 2015.

[40] Zhang Xianyi, Wang Qian, and Zaheer Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2014.

[41] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[42] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. *SIGPLAN Not.*, 53(3):45–56, March 2018.

[43] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax tree for programming language understanding and representation: How far are we?, 2023.

[44] AWS What is New. Introducing amazon ec2 spot placement score, 2021.

[45] Azure. az compute-diagnostic spot-placement-recommender, 2024.

[46] Sungkyu Cheon, Kyumin Kim, Kyunghwan Kim, Moohyun Song, and Kyungyong Lee. Multi-node spot instances availability score collection system. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)*. ACM, July 2025.

[47] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. *SIGPLAN Not.*, 52(4):645–659, apr 2017.

[48] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: The case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 73–87, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Mario Juric, Steven Stetzler, and Colin T. Slater. Checkpoint, restore, and live migration for science platforms, 2021.

[50] Renato L. F. Cunha, Lucas C. Villa Real, Renan Souza, Bruno Silva, and Marco A. S. Netto. Context-aware execution migration tool for data science jupyter notebooks on hybrid clouds. In *2021 IEEE 17th International Conference on eScience (eScience)*, pages 30–39, 2021.

[51] D Fraser, E Horner, J Jeronen, and P Massot. Pyan3: Offline call graph generator for python 3, 2020.

[52] Gharib Gharibi, Rakan Alanazi, and Yugyung Lee. Automatic hierarchical clustering of static call graphs for program comprehension. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4016–4025, 2018.

Soohyuk Lee is a B.S. student in the Department of Computer Science at Kookmin University, South Korea. His research interests are in the area of cloud computing and computer networking.

Junho Lim received the M.S. degree from Kookmin University, Seoul, South Korea. He is currently with PIOLINK, Inc., Seoul, South Korea. His professional interests are in the areas of cloud computing and network security.

Kyungyong Lee is an associate professor in the department of Data Science at Hanyang University, Seoul, South Korea. His research topic covers optimizing cloud computing environment for diverse workloads. He received the Ph. D. degree in the Department of Electrical and Computer Engineering at the University of Florida.