Workload-Aware Live Migratable Cloud Instance Detector

Junho Lim
lim-junho@kookmin.ac.kr
Computer Science. Kookmin Univ.
Seoul, South Korea

KyungHwan Kim
bryan9801@kookmin.ac.kr
Computer Science. Kookmin Univ.
Seoul, South Korea

Kyungyong Lee
leeky@kookmin.ac.kr
Computer Science. Kookmin Univ.
Seoul, South Korea

***Abstract*—Cloud computing provides a variety of distinct computing resources on demand. Supporting live migration in the cloud can be beneficial to dynamically build a reliable and cost-optimal environment, especially when using spot instances. Users can apply the process of live migration technology using the Checkpoint/Restore In Userspace (CRIU) to achieve the goal. Due to the nature of live migration, ensuring the compatibility of the central processing unit (CPU) features between the source and target hosts is crucial for flawsless execution after migration. To detect migratable instances precisely while lowering false-negative detection on the cloud-scale, we propose a workload-aware migratable instance detector. Unlike the implementation of the CRIU compatibility checking algorithm, which audits the source and target host CPU features, the proposed system thoroughly investigates instructions used in a migrating process to consider CPU features that are actually in use. With a thorough evaluation under various workloads, we demonstrate that the proposed system improves the recall of migratable instance detection over $5\times$ compared to the default CRIU implementation with 100% detection accuracy. To demonstrate its practicability, we apply it to the spot-instance environment, revealing that it can improve the median cost savings by 16% and the interruption ratio by 15% for quarter cases.**

***Index Terms*—Migration, ISA, Cloud, Debugging**

I. Introduction

The evolution of cloud computing changes the way computing resources are used in various fields. Notable cloud computing characteristics that lead to broad adoption are elastic resource usage, on-demand billing, and a variety of instance types that users can choose. With various instance types, applying the live migration of applications can significantly enhance efficiency as the resource requirements of the application change. Live migration of applications in the process or container layer can be performed without the support of cloud service providers using a process migration system, such as CRIU [1].

When migrating applications between various instance types with distinct CPU features, flawless operation after migration is crucial. One of the safest ways to achieve the goal is to select a migration target instance that supports all features supported by a migration source instance, and it can be ensured by checking the advertised CPU features of host machines. The current implementation of the CRIU compatibility checking module adopts this approach and compares the CPU features of the source and target instance before starting a migration. However, a workload generally does not use all CPU features supported in the source instance, and considering the unused features in the migratable instance detection can negatively affect the number of feasible target instances for migration, which we referred to as false-negative migratable instance detection that results in low recall.

To overcome these limitations, we propose a workload-aware migratable instance detection algorithm. Unlike the simple detection algorithm adopted by CRIU, the proposed system first extracts instructions that are highly likely to be executed. The extracted instructions are transformed into CPU features to generate a workload-to-CPU feature map. The map is then matched to the CPU feature maps extracted from various cloud instance types to identify migratable instance types. In the proposed system, precise workload instructions analysis is crucial, and we propose two heuristics, *text-segment full scan* and *execution path tracking*, which have trade-offs between recall, operation overhead, and the completeness of the feature extraction.

To demonstrate efficiency using the proposed system, we collected the CPU features of 450 unique instance types provided by Amazon Web Services (AWS). We chose seven real-world workloads that use different CPU features. The results of the experiment reveal that the proposed algorithm can identify a migratable instance with 100% precision. Compared to the CRIU detection algorithm, the proposed method has a $5\times$ higher recall value, which greatly enlarges the target node pool for migration. The effectiveness of the proposed system is further demonstrated by adopting it in a spot-instance environment in the cloud by demonstrating that the proposed system improves the median cost savings by 16% with improved reliability.

The major contributions of this paper are as follows.

- Proposing heuristics to extract instructions of workload and matching them to corresponding CPU features
- Conducting thorough analysis of various issues when analyzing and matching instructions to CPU features
- Implementing the system to demonstrate the effectiveness. To the authors' best knowledge, this is the first work to enhance the migratable instance detection in the cloud-scale.

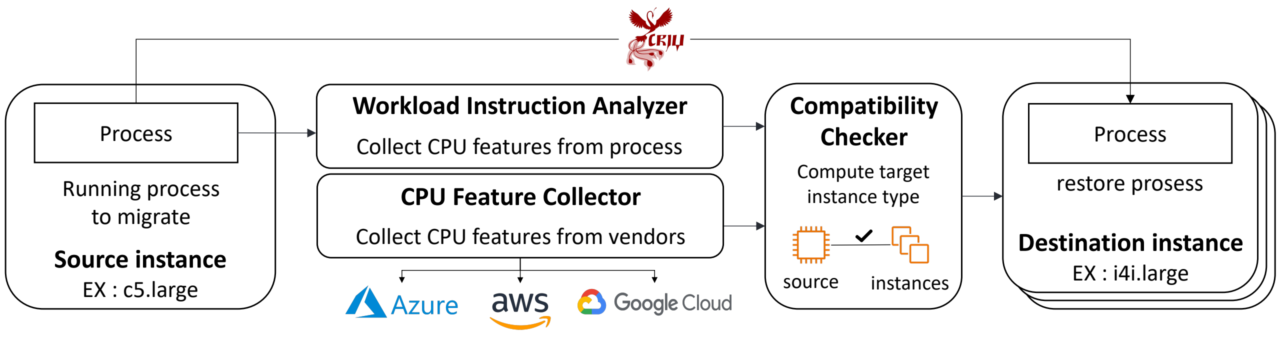


Fig. 1: Components of the proposed system to find a set of cloud instances to which can migrate

II. Runtime Migration on Public Cloud

Runtime migration allows one to change an application hosting machine to another host. Live migration, also known as hot migration, keeps a migration target running while preserving volatile memory content and other system statuses, such as open file descriptors, sockets, and files. Migration can be applied to achieve various goals, such as load balancing of many computing resources and continuous operations even in a disaster [2]. Migration can occur between virtual machines [3], containers [4], [5], or processes [1], [6]. In a public cloud computing service, virtual machine migration is not publicly supported due to its implementation complexity. It is intended to be used internally by a service vendor to provide continuous service during machine maintenance. Instead, users can initiate migration in the context of a container or process that runs on top of a virtualized instance.

A. Live Migration Process

The CRIU [1] is widely used to migrate a process or container while keeping an application running without losing volatile memory content. The CRIU migratable objects include virtual memory mappings, memory and register content, file descriptors, sockets, pstrees, and many others. The sequence of CRIU is divided into checkpoint and restore operations.

The checkpoint step freezes the target process for migratin using a *cgroup freezer* feature to avoid changes while dumping the process content. It recursively scans the */proc/$pid/* directory to extract process resources. Then, using a *ptrace* system call, it injects a parasite code that allows the CRIU routine to be executed in the target process to dump the memory content. The checkpoint object is stored using a protobuf [7] format that can be restored later.

In the restore step, CRIU analyzes the checkpointed file to understand which resources are shared by the migrated process. During this process, shared resources are identified, such as file handles, network connections, and shared memory areas. To create the necessary new address space for restoration, CRIU invokes the *fork* function multiple times to prepare a process tree for the restoration. This prepared process tree replicates the original process state based on memory mappings, file descriptors, socket states, and other information stored in the checkpoint file.

B. Ensuring Compatibility Based on CPU Features

The target machine must support all CPU features used by the migrating process to ensure compatibility during migration. Suppose a restored process attempts to execute an instruction that the migration target host CPU cannot interpret. In that case, the CPU generates a `invalid opcode error`, leading to the abnormal termination of the process. The most intuitive method to ensure compatibility is to choose a migration target instance that supports all CPU features supported by a source instance. The current CRIU implementation adopts this approach. Although it provides a simple implementation, it can result in many false-negative migratable instance detections, which lowers the number of migratable instances, because most applications do not use all the features provided by a source host CPU.

1) CPU Dispatching: The CPU dispatching technique determines which version of the source code to execute based on the CPU features of the underlying hardware to make the application source code compatible with multiple hardware types. As an implementation of CPU dispatching, **function multiversioning** [8], [9] allows developers to implement multiple versions of a single function, each optimized for different CPU features. A compiler produces multiple versions of the implementation. In the runtime, the GNU indirect function (*IFunc*) resolver calls *__cpu_indicator_init* and inspects CPU features.

One caveat when using function multiversioning is that it dispatches a function only at the start of a process. Although this approach minimizes the overhead of dispatching every time a function is executed, it can cause problems during migration. If a process is dispatched to a specific CPU feature and migrated to a host machine that does not support the feature, it can potentially cause a problem.

C. Live Migration for Cloud Spot Instances

Public cloud service providers are equipped with abundant computing resources to meet the dynamic demand for them. Such abundant resources inherently result in a surplus when demand is low, and many public cloud vendors provide the remaining computing resources at a lower price, which is generally referred to as a spot instance. In response to the discounted price, a spot instance can be terminated whenever computing resource demand increases, and spot-instance users should be prepared for a sudden node interruption.

In the literature, considerable research work has been conducted to deal with spot-instance interruptions by setting a proper spot-instance bidding price [10], [11], using the checkpointing feature inherent to a specific application to restart from a checkpoint status when an interruption occurs [12], [13], or a mixture of spot and on-demand instances to achieve cost efficiency and reliability [14], [15].

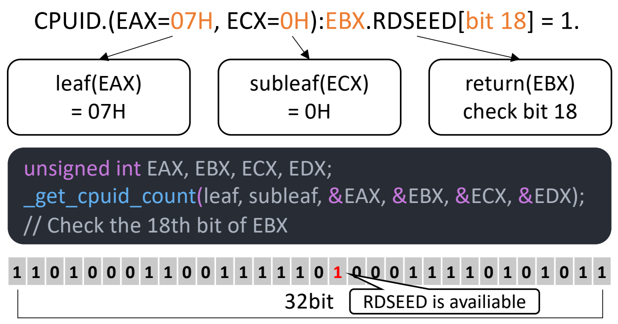
To the best of the authors' knowledge, no research has attempted to apply the live migration process on an interrupted spot instance. However, it could be a viable approach to deal with an interruption event, even for applications that do not natively support checkpointing. The cost savings and availability of a spot instance vary significantly for various instance types globally [16]. Among many possible candidates for migration, it is crucial to guarantee the compatibility between the source and target instance types. The live migration compatibility checking heuristic proposed in this paper facilitates ensuring flawless recovery from spot-instance interruption while improving cost savings and reliability after migration.

III. Workload-Aware Migratable Instance Detector

Compatibility checking for a process live migration based on CPU features, as the current implementation of CRIU does, might unnecessarily filter out feasible target instances even when missing CPU features in a target are not used in a workload. To avoid such a false-negative migration feasibility detection and widen the migration candidate node pool in the cloud, we propose a workload-aware migratable instance detector. Figure 1 illustrates the components of the proposed system. The *CPU feature collector* (Section III-A) is responsible for gathering CPU features of various cloud instances. The *workload instruction analyzer* (Section III-B) investigates the process for migration to detect core operations in which compatibility should be guaranteed on a new host. The *compatibility checker* (Section III-C) determines whether a process can be migratable to a specific instance type.

A. CPU Feature Collector

Various instance types, even a few hundred with unique CPU features, are offered by public cloud service providers. The CPU feature collector module gathers CPU features of unique instance types off-line to make a prompt decision about migratable instance types. In systems with X86 architecture, CPU features can be extracted using the CPUID instruction [17], whereas specific system registers are referenced for this purpose in ARM systems. This paper primarily focuses on the feasibility of real-time migration between x86 (x64) Instruction Set Architecture (ISA) systems, as most current cloud instances fall into this category. The CPUID in X86 provides detailed information about the CPU model, family, and supported instruction sets, depending on the given CPUID leaf (EAX) and subleaf (ECX) arguments. The CPU feature collector executes the CPUID instruction using the



```
unsigned int EAX, EBX, ECX, EDX;
_get_cpuid_count(leaf, subleaf, &EAX, &EBX, &ECX, &EDX);
// Check the 18th bit of EBX
```

Fig. 2: Using CPUID to check whether a CPU feature is supported by a host machine

__get_cpuid_count method provided in the GNU C Library (glibc). The acquired CPU information is stored in various registers (EAX, EBX, ECX, and EDX), containing data based on the values set for the leaf and subleaf. Figure 2 depicts an example of checking whether a host machine supports the read random seed (RDSEED) CPU feature. According to the X86 architecture manual, the feature is defined as *CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 1*. When calling *__get_cpuid_count*, one must set the leaf (EAX) to seven and the subleaf (ECX) to zero. The feature support is determined by checking whether the 18th bit of the EBX register is set to one.

B. Workload Instruction Analyzer

The proposed system analyzes the operations of a workload with respect to the CPU features that the workload needs in a new host. To extract the CPU features, we propose two methods: the *text-segment full scan* and *execution path tracking*. These methods analyze the text section of the process memory, which contains all instructions and function calls from an executable binary file and the shared libraries that might be executed during the program runtime. For the extracted operations, the system applies the Intel X86 Encoder Decoder (XED), which can encode and decode details of X86 instructions, to identify the mapping of an instruction to a CPU feature.

1) Text Segment Full Scan: The text-segment full scan decodes all CPU instructions loaded into the process memory using XED to extract the corresponding CPU features. Using the GNU debugger (GDB) and Capstone disassembler, we implement it to analyze the set of CPU instructions in the process memory. GDB monitors the internal activities of running programs and collects addresses of the text segment, whereas the Capstone disassembler is responsible for disassembling these addresses.

The algorithm 1 illustrates the operation sequence of the text-segment full scan. First, a migration target process is loaded into GDB, and the start and end addresses of each text section are collected (Lines 1-2). All operations in the text section are disassembled using the Capstone disassembler, and the operations in the text segment are collected (Lines 3-

Algorithm 1: The sequence of text segment full scan

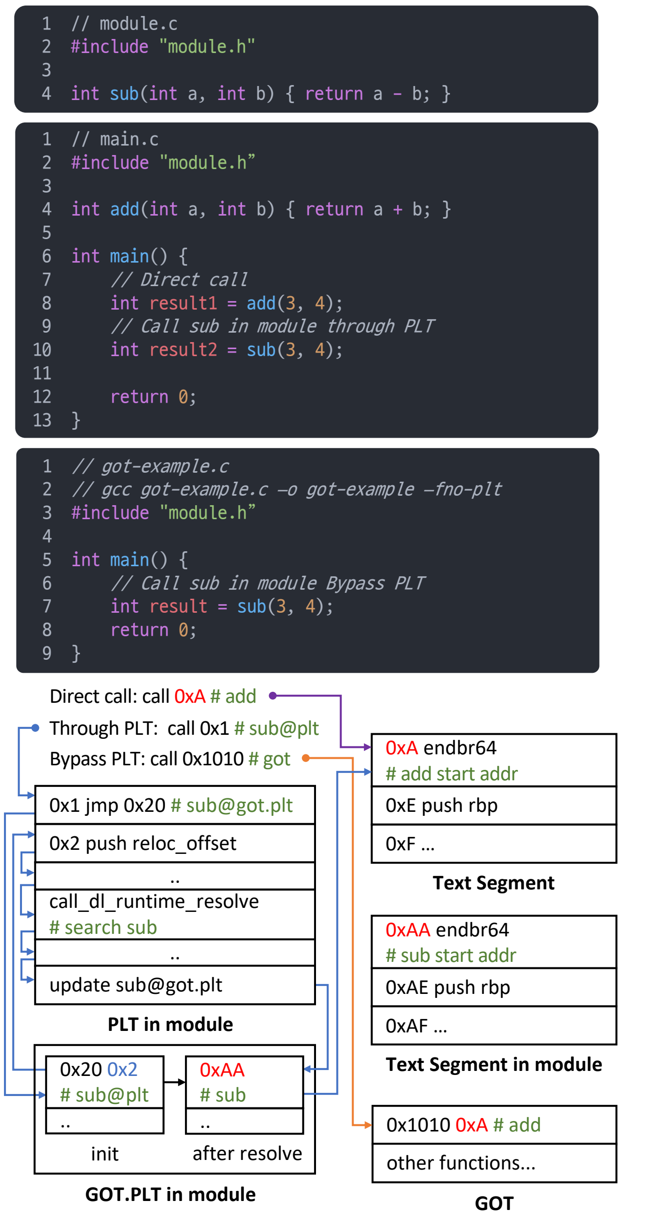
```
   Result: A set of CPU features for an input workload
1  Inputs: P: Process
2  for text_section_addr as T in P do
3      for start_addr, end_addr in T do
4          instructions ← $disas(start\_addr, end\_addr)$
5      end
6      instructions ← $instructions.deduplicate()$
7      for instruction as I in instructions do
8          CPU_feature ← $xed\_decode(I)$
9          workload_cpu_features.append(CPU_feature)
10     end
11 end
12 workload_cpu_features.deduplicate() return
     workload_cpu_features
```

5). Line 6 deduplicates the operations, and the deduplicated disassembled instructions are decoded to the corresponding CPU features using XED. The identified CPU features are stored (Lines 7-10), and after scanning all text segments, duplicated CPU features are removed, returning the result (Lines 12-13).

The text-segment full-scan approach scans the entire text segment and can incur significant overhead for disassembling and CPU feature decoding. It also checks all CPU features that can be referenced from a workload that is not executed due to CPU dispatching. For example, the function `strlen` in GLIBC is implemented in several optimized versions, such as AVX2 and AVX512. The full-scan approach does not consider these details. It tracks all versions of `strlen` implementation in the text segment that may result in false-negative detection for a migratable instance, but it can track all the possible instructions a process might execute.

2) Execution Path Tracking: To avoid unnecessary scanning of the entire text segment, we propose an *execution-path-tracking* heuristic to extract CPU features actually used by the process in a best-effort manner. The proposed method tracks every possible execution path from the entry point to achieve this goal. The method follows *call* and *jmp* related operations to track the code execution path. A function can be called in three ways: a direct call, a call through the Procedure Linkage Table (PLT), and a PLT bypass. We describe how various calling mechanisms are tracked.

Direct Call: This method is predominantly used in static linking, where all functions and libraries are included in the executable file at compile time. In the direct call method, a compiler knows the memory address of the function beforehand and it can call the function through its absolute address. This method allows for fast function execution because it does not require additional address resolution at runtime. Even in a dynamic linking environment, the user-written code or functions defined within the program are directly included in the executable, and the direct call method is used. Figure 3 shows a sample code, its corresponding GDB output, and the



```
1 // module.c
2 #include "module.h"
3
4 int sub(int a, int b) { return a - b; }
```

```
1  // main.c
2  #include "module.h"
3
4  int add(int a, int b) { return a + b; }
5
6  int main() {
7      // Direct call
8      int result1 = add(3, 4);
9      // Call sub in module through PLT
10     int result2 = sub(3, 4);
11
12     return 0;
13 }
```

```
1 // got-example.c
2 // gcc got-example.c -o got-example -fno-plt
3 #include "module.h"
4
5 int main() {
6     // Call sub in module Bypass PLT
7     int result = sub(3, 4);
8     return 0;
9 }
```

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

memory layout. The *add* method in *main.c* of line number 4 is called in the main function line 8, using a direct call method. In the GDB output, the function is called referencing the absolute address of *add* function.

Call through PLT: This method is employed in dynamic linking where external library functions are not directly included in the executable but are loaded into memory by the operating system at runtime. Consequently, the compiler does not know the exact memory address of the function and compiles the function to be called referencing the PLT. The PLT is a section within an executable file, designed to mediate calls to external functions and libraries. The PLT is engineered to reference the GOT.PLT section in order to facilitate function calls, where the GOT.PLT is part of the Global Offset Table (GOT) [18], [19]. The GOT primarily serves as a table for storing addresses of global variables and functions from dynamically linked libraries, and the GOT.PLT specifically manages addresses of functions. Initially, the GOT.PLT points to a routine to locate the address of library functions, and this routine is activated when a specific function is called for the first time. During the initial call to a specific function, the PLT routine calculates the actual address of the function and updates the GOT.PLT. Subsequent calls to the function do not require a lookup of the symbol, as they can directly reference the GOT.PLT for the function call, allowing the dynamic linker to find and link the function address at runtime [20].

The method *Through PLT* in Figure 3 illustrates the initial call to a function via the *PLT*. The function *sub* defined in *module.c* is called in the *main* function of *main.c* in line 10, where the *sub* function is assumed to be loaded from an external library by dynamic linking.

Initially, the address of the *sub* function is sought by calling the address of *sub@plt* (0x1). The *sub@plt* references the *GOT.PLT*, a table updated at runtime and accessible by the *PLT*, to perform a *jmp* to the address of the target function. *GOT.PLT* is initially not updated with the address of the *sub* function. Therefore, the address of *sub* must be found and updated in *GOT.PLT*. Initially, *GOT.PLT* points to the next item of *sub@plt* (0x2), following the execution flow, it locates the address of the function to call from the library. At this step, *_dl_runtime_resolve* is called, locating the address of *sub* and recording it in *GOT.PLT*. Subsequent calls to the function can directly reference *sub@GOT.PLT*, eliminating the necessity to locate the address again.

This method enables library sharing, allowing multiple programs to share the same library code, offering the advantage of memory savings. However, this can incur address-resolution overhead. In summary, functions embedded in the binary are called through their absolute addresses determined at compile time (Direct Call), whereas functions from external libraries are called at runtime through addresses found in the PLT.

PLT Bypass: Some compilers provide techniques to bypass the PLT, accessing the GOT directly during function calls to optimize performance. This approach uses the addresses already updated in GOT at runtime initialization, reducing the overhead of additional jumps and lookups during function calls. Figure 3 shows an example of PLT bypass. In the *got-example.c* source code, which is the third code snippet, it utilizes the *GCC -fno-plt* compile option to bypass the PLT when calling the *sub* function. By using the option, it directly points to the real function address located in GOT.

In the current implementation of the execution-path-tracking module, it supports all the above-mentioned function-calling mechanisms.

Algorithm 2 explains the procedure of execution path tracking. On lines 1 to 3, a process is loaded into GDB. After loading, the starting address of the primary function is set as the starting point and added to the tracking list. In line 5, functions within the tracking list are disassembled, and the results are parsed into individual instructions. Lines 6 and 7 check whether an instruction is a branch instruction. In the current implementation, the algorithm tracks 24 branch instructions for the *call* or *jmp* related ones. On lines 8 to 10, if the destination address of a branch instruction is trackable, it is added to the trackable list. To decide whether a function is trackable, we use disassembly output metadata provided by GDB. If GDB can find a symbol for a branch target function, it includes the metadata of the function as comments, such as the function's name and address. The direct call mechanism is trackable using GDB output by simply following the address field value, which points the text segment address. When a call is made by referencing the PLT, the target function can be located within the GOT.PLT, and it is marked as trackable. Similarly, if a call is facilitated through the GOT bypassing PLT, the target function can also be identified within the GOT. The differentiation between these calling methods relies on the annotations provided by GDB. Beyond these instances, any calls are considered untraceable. This encompasses scenarios where symbols have been eliminated due to optimization or security reasons, cases where the symbols have yet to be loaded, or a function address is referenced from a register value, which is implemented as calling by function pointer.

Algorithm 2: Execution Path Tracking

```
Result: CPU features from workload
1  Inputs:    $P$: Process
2  main_addr ← $get\_main\_addr()$
3  tracking_list.append(main_addr)
4  for func_start_addr as A in tracking_list do
5      instructions ← $disas\_func(A)$
6      for instruction as I in instructions do
7          if I == branch instruction then
8              if is_trackable (I) then
9                  tracking_list.append(get_func_addr())
10             end
11         end
12         workload_cpu_features.append(xed_decode(I))
13     end
14     workload_cpu_features.deduplicate()
15 end
16 return workload_cpu_features
```

Line 12 decodes all disassembled instructions using XED to collect the corresponding CPU features of each instruction. Specifically, the bytecode of each instruction is passed to XED. Based on the opcode, operands, prefixes, and other elements that can be extracted from the bytecode, XED determines which CPU feature a given instruction belongs to. After all possible paths are tracked, on lines 14 to 16, the duplicated CPU features are removed, and the final set of features is returned. The execution path tracking process follows the sequence of function calls; thus, it excludes all functions that are not executed, although they are present in memory.

The CPU dispatching of function multiversioning happens once in the program startup, and only the dispatched implementation is tracked during execution path tracking, implying that an implementation dispatched in a migrating source instance remains intact in the target instance after a migration. Thus, an incompatibility problem can be detected using the proposed method. However, suppose that a user implements a custom CPU dispatching module using a conditional statement. In that case, all implementations can be tracked, and it can still result in a false-negative migratable instance detection.

In summary, the proposed execution path tracking heuristic focuses solely on tracking branches within the flow of executed function calls, thereby extracting CPU features used in functions that are actually called, while excluding functions not executed in the code path. Compared to the *text-segment full-scan* method, the scope of the code subject to tracking is significantly reduced and prevents unnecessary analysis of irrelevant codes.

C. Compatibility Checker

The compatibility checker module determines the compatibility of the source and destination hosts based on the CPU features of a process extracted from the workload instruction analyzer and the CPU feature collector. Compatibility checking is conducted by comparing whether the CPU features used in the migrating workload on the source hosts are present on the destination host. Although this evaluation method is similar to the default CRIU implementation, it focuses on the CPU features that a target workload uses. Thus, it is expected to lower the chance of false-negative migratable instance detection.

IV. Issues During Development

When analyzing the CPU features used by a process, multiple issues arise as the CPU hardware evolves, and we summarize the issues and how we handle them.

A. Handling TSX Issue

Transactional synchronization extensions (TSX) [21], [22] are designed to optimize parallel processing in multicore processors. Support for TSX is identified using the CPU features of restricted transactional memory (RTM) and hardware lock elision (HLE). However, the TSX method has been identified to be vulnerable to side-channel attacks, raising significant security concerns [23], [24], particularly with regard to the potential leakage of sensitive information by observing memory access patterns using TSX. Therefore, the TSX feature may have been disabled in most CPUs. In such cases, the RTM and HLE features are reported as zero through CPUID, and the core TSX commands, such as *xbegin* and *xend*, which open and close a transactional memory region, are not available. However, we discovered that the *xtest* command, which checks whether the code is executed in a transactional region, may still function on CPUs where TSX is disabled, which can cause false-negative detection for a workload that uses only the *xtest* command in a migration target instance without the RTM and HLE features.

The CPU feature collector module additionally checks whether a host machine can execute the *xtest* command regardless of TSX support to address this problem. The compatibility checker module checks whether a workload requires both the RTM and the HLE features or requires the ability to execute the *xtest* command using the additional feature.

B. Limitations of Execution Path Tracking

The execution-path-tracking module cannot track a function called with a pointer. For example, if a function is called using the `call eax` instruction. General-purpose registers such as *EAX* and *EBX* undergo continuous value changes during the execution of the process. Consequently, to accurately track calls made through registers, it is imperative to read the register values at the exact moment these calls occur. This requirement entails continuous monitoring throughout the entire life cycle of a process, leading to significant overhead. Therefore, the proposed system acknowledges the limitations in tracking calls made via register references, and the impact of this limitation is thoroughly addressed in the evaluation section.

Lazy loading allows a process to load a specific library when required, even after a program is started [25], [26]. Using the *dlopen()* method, one can quickly implement the feature. This approach is effective in reducing the initial load times and optimizing memory usage, particularly by facilitating the efficient use of system resources in large-scale or resource-intensive applications. However, when a library has not been loaded, its symbols do not exist, making it impossible for the execution-path-tracking module to trace symbols from an unloaded library.

C. Constraints in Binding Methods

Lazy binding is a technique used for performance optimization, in which the function address is not resolved until the function is called. In other words, in dynamic linking, the PLT does not point to the actual function address until the first call to the function occurs. This method reduces initial loading time and optimizes memory usage during execution, making it particularly beneficial for complex programs that use large libraries. The execution-path-tracking module operates by identifying the function addresses. If the address of the function to be called is not resolved at the time of tracking, it is not included in the tracking list.

Unlike lazy binding, the now binding approach resolves the addresses of all external symbols at the beginning. With now binding, execution path tracking can trace all function addresses even at the beginning of program execution, enabling tracking at any point during the process runtime. Due to the limited function address identification of lazy binding, the current system supports the now binding approach.

V. Evaluation

This section evaluates the proposed migration compatibility checker modules from the perspective of precision and recall of the compatibility decision with an analysis of the overhead to run the system for various workloads. With the evaluation, we aim to answer the following questions.

1) **RQ-1**: Do the proposed text-segment full scan and execution-path-tracking modules provide better performance than the traditional CPU compatibility checking method provided by CRIU?
2) **RQ-2** : Is the overall overhead to perform instruction analysis in the proposed modules tolerable to be applied practically for a real application?
3) **RQ-3** : When the proposed compatibility check module is applied in a real cloud environment, how much monetary benefits are expected, especially when using volatile spot instances?

A. *Environment Setup*

We collected the CPU feature maps of 450 unique AWS EC2 X86 instance types using the proposed CPU feature collector module. Verification of directional migration compatibility for all pair combinations of n instance types requires $\frac{n\times(n-1)}{2}$ experiments, which is over 100,000 for 450 instance types, which can be prohibitive in time and cost. To minimize the overhead, we grouped instance types according to CPU features, resulting in 27 unique groups. Within each group, we performed all-to-all migration compatibility experiments to ensure that there were no problems in the migration in the same group. For inter-group migration, we chose one instance type in each group, preferring a lower price, and migrations were conducted between the selected instance types, resulting in 702 (27×26) directional, unique migrations for each workload.

To check the migration compatibility using a realistic workload, we used three general workloads and four synthetically generated workloads to employ specific CPU features. Commonly used workloads includes matrix multiplication using OpenBLAS [27] which is a core kernel of various deep neural network and machine learning algorithms. We used the an in-memory key-value storage Redis while continuously performing read and write operations. As the training workload for a machine learning model, we used extreme gradient boosting (XGBoost) [28] to train a classification model using a gradient boosting algorithm with the MNIST dataset. The XGBoost workload is implemented in the CPP, whereas the other workloads were written in the C language. Special-purpose workloads include OpenSSL-based Rivest, Shamir, Adleman (RSA) encryption and decryption using the Multi-Precision Add-Carry Instruction Extensions (ADX) feature optimized for large integer operations. Another workload is a memory protection example using the protection keys for user-mode pages (PKU) key-based memory protection feature [29], [30], a random number generation example using the RDSEED feature, and an OpenSSL-based hashing example using the secure hash algorithm (SHA) feature. The special-purpose workloads are marked as adx, pku, rdseed, and sha, respectively.

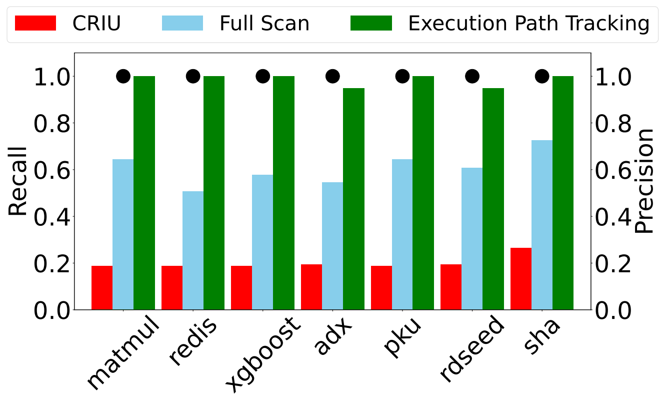


Fig. 4: Comparison of CRIU, text segment full scan, and execution path tracking precision and recall

B. *Precision and Recall*

We first aimed to answer **RQ-1** regarding migration compatibility detection performance. In the evaluation, we used precision and recall as performance metrics. The precision measures the accuracy of detection. It measures among the predicted migratable instance pairs, how many pairs are truly migratable. The recall measures how many migratable pairs are detected out of the actually migratable instance pairs. Precision focuses on the evaluation of true-positive and false-positive predictions, and the recall focuses on the evaluation of true-positive and false-negative detections. Lower precision implies migration failure due to crashing, and a lower recall implies missing many feasible instance types as migration targets.

Figure 4 compares the recall and precision of various migratable instance detection algorithms. In the figure, the horizontal axis indicates various workloads. The primary vertical axis displays the recall, and the secondary vertical axis presents the precision value. The recall value of various detection algorithms is presented using bars marked with distinct cross-line, upper-right diagonal, and upper-left diagonal patterns that mark CRIU, text-segment full scan, and execution path tracking, respectively. For the 4,194 migration experiments, all three detection algorithms have a precision score of 1.0, which is marked with circle markers; thus, there was no migration failure if a detection algorithm predicted migration success.

Recall indicates quite different values for distinct algorithms. The default CRIU implementation has the lowest recall value, followed by the text-segment full scan. Execution path tracking has the best recall value. The CRIU detection algorithm does not consider the workload characteristics, and many possible migration target hosts were excluded. The text-segment full-scan method reflects the workload characteristics, and its recall is better than the default CRIU. However, it still filters out many possible instance types from the feasible list. Finally, the execution-path-tracking module could precisely detect all feasible migration target instances for most workloads. It could not achieve the recall of 1.0 for

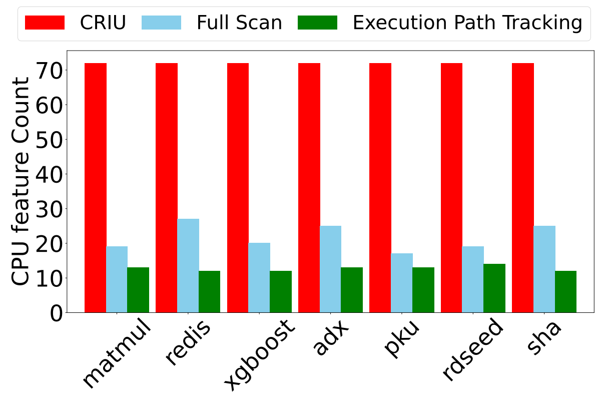


Fig. 5: Comparing the number of CPU features considered during migratable instance detection

Workload	Traceable	Untraceable	Ratio
matmul	10,899	184	0.017
redis	32,547	372	0.011
xgboost	18,521	300	0.016
adx	11,765	184	0.016
pku	10,851	172	0.016
rdseed	10,262	169	0.016
sha	10,274	168	0.016

TABLE I: The number of traceable and non-traceable function call instructions when using the execution path tracking module

the RSA and RDSEED workloads. For both these workloads, we observed that they are considered to use the CPU feature of *ADOX_ADCX*. However, the workloads could be executed on some instance types without the feature. We could not generalize the executable and inexecutable cases with respect to the feature, and we leave the case as a missing instance detection for the execution-path-tracking module.

Figure 5 presents the number of CPU features considered when making a migration detection on the vertical axis. The horizontal axis and bar notation are the same as in Figure 4. The default CRIU implementation checks 72 unique CPU features regardless of the workload. The proposed workload-aware detection mechanism presents a lower number of considered CPU features. Filtering out irrelevant CPU features during migratable instance detection could improve the recall of the proposed method.

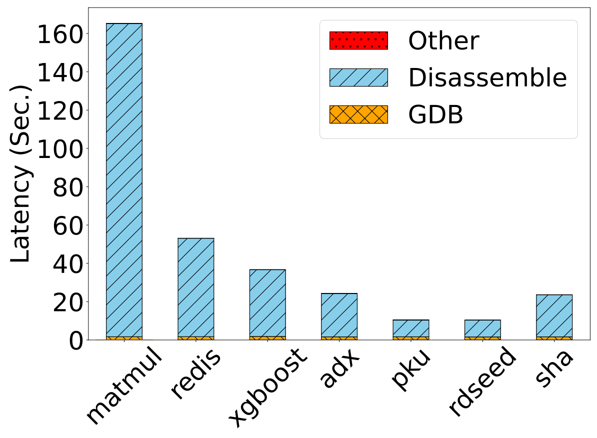
Table I presents the number of traceable and untraceable functional call instructions and the ratio of untraceable to traceable when using execution path tracking. Unlike the text-segment full-scan method, execution path tracking may lead to tracking omissions in certain situations, such as calling a 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 miss, 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, less than 2% 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 deal with the untraceable function problem. If the tracking miss is not tolerable for any reason, one can apply the text-segment full scan heuristic whose recall is worse than the execution path tracking but better than the default CRIU implementation.

C. Proposed System Operation Overhead

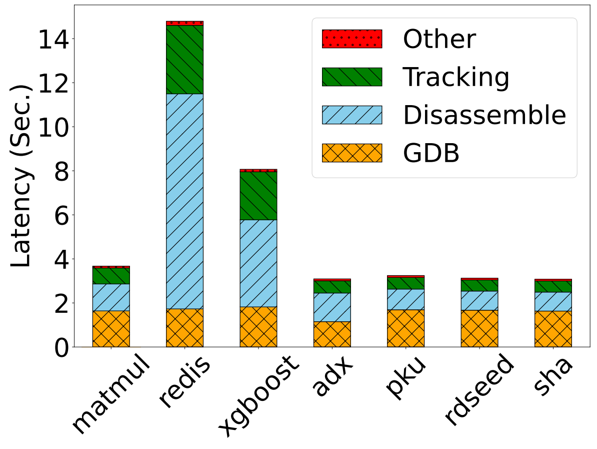
This section answers **RQ-2**, related to the additional overhead to execute the proposed system to enhance the recall of migratable instance detection. Figure 6 presents the overhead when analyzing the workload instructions and the corresponding CPU features to understand the overhead quantitatively. Figures 6a and 6b illustrate the analysis time of the text-segment full scan and execution path tracking methods, respectively. The primary vertical axis displays the latency, and the corresponding value is presented in a stacked bar format. The *GDB* represents the time taken for a workload to load into the GDB. *Disassemble* is the time to disassemble the text section or a function. *Tracking* refers to the time to follow the execution path to locate the address of the next target function. *Other* comprises the accumulated latency of various minor factors, such as obtaining the address of the text sections, exporting the tracking results to a file, and decoding instructions using XED. The overhead measurement was performed on AWS *c5.large* instances.

For the text-segment full scan heuristic, disassembling the entire text segment takes the majority of the time. For most workloads, it took about 1.5 seconds to load a process to GDB for execution path tracking in Figure 6b. The disassembling instructions took the longest time, followed by the tracking time. The real-world workload took longer than the synthetically generated workloads due to the large source code size. One interesting observation is that the matrix multiplication with OpenBlas took the longest time for the text-segment full scan but decreased significantly for the execution-path-tracking method. A thorough analysis reveals that the size of the OpenBlas library is quite large compared to other workloads, and the text-segment full scan should scan the entire library. However, the matrix multiplication workload accesses only a fraction of the library functions, resulting in a significant latency reduction in the analysis time.

To compare the number of disassembled instructions for various workloads, Table II compares the metric between the text-segment full scan and execution path tracking methods. The last column, *Ratio*, is calculated as the ratio of the number of disassembled instructions of execution path tracking to the text-segment full scan. The lower ratio implies that execution path tracking filters out more unnecessary functions. For matrix multiplication, the number of disassembled instructions decreases significantly, about 99.4%. For the text-segment full scan, it is observed that the overhead increases linearly with the size of the binary or the amount of source code including



(a) Text Segment Full Scan overhead



(b) Execution Path Tracking overhead

Fig. 6: The operation overhead of runtime process analyzer

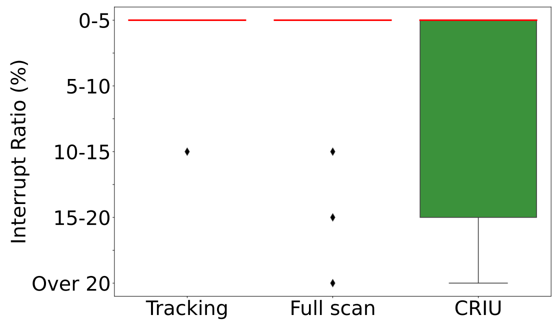
Workload	Text Segment Full Scan	Exec. Path Tracking	Ratio
matmul	8,568,489	52,335	0.006
redis	2,557,052	154,053	0.060
xgboost	1,778,354	90,316	0.051
adx	1,072,844	56,511	0.053
pku	435,113	52,565	0.12
rdseed	435,085	49,523	0.114
sha	1,072,779	49,584	0.046

TABLE II: The number of disassembled instructions for the text segment full scan and the execution path tracking

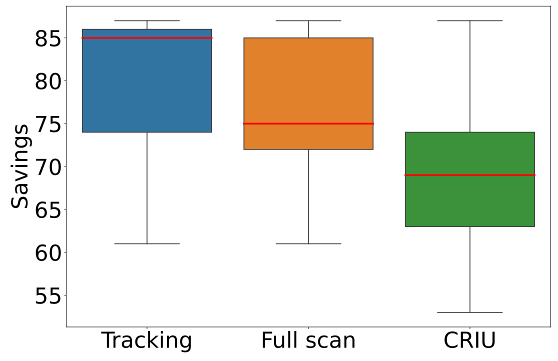
library. The overhead of execution path tracking increases with the number of functions actually called rather than with the size of the binary. Although the overhead might increase indefinitely in both cases as the program size increases, the deduplication step to filter out the already checked functions can help decrease the overhead. Furthermore, the efficiency of execution path tracking to analyze functions that are likely to be executed can significantly lower the overhead.

D. Integration with Spot Instances for Cost Savings

To answer **RQ-3** regarding the practicality of the proposed migratable instance detection algorithm, we present



(a) Frequency of interruption



(b) Cost savings

Fig. 7: Enhancements in the reliability and cost savings when the proposed system is applied in a spot instance environment

the enhanced reliability and cost savings when the proposed algorithm is applied in a spot-instance environment.

The proposed workload-aware migratable instance detection algorithms increase the number of candidate instances to which an interrupted spot instance can migrate. For example, a matrix multiplication workload using the OpenBlas library in an AWS *c6a.24xlarge* instance type can be migrated to seven distinct instance types when using the CRIU implementation. When using the proposed execution path tracking method, the number of migratable instances increases to 449, which can provide great flexibility when choosing target instances. Many cloud vendors that support spot-instance services provide various datasets, such as cost savings, interruption ratios, or instant availability [16], [31]. The effectiveness of the proposed algorithms is quantitatively evaluated by measuring cost savings and interrupt ratio datasets across many instance types, due to more migratable instances.

Figure 7 presents the interruption ratio and cost savings using a box-whisker format with different migratable instance detection algorithms depicted on the horizontal axis. Compatibility checks were performed for all workloads by having 27 groups with unique CPU features as migration source instances. The target instances were selected from among 449 instance types of the same size. We assumed selection of an instance type offering the lowest interruption ratio and cost savings from the migratable instances identified by various algorithms. To obtain the timely spot instance and interruption ratio data, we utilized SpotLake datasets. [16].

Figure 7a presents the improvement in the interrupt ratio from the expansion of the migration candidate nodes using the proposed algorithm. 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 prior to the 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%. In the median cases, the three detection mechanisms could select instance types whose interrupt ratio is less than 5%. However, when using the Execution Path Tracking mechanism, approximately 7% of the selected target instances had a termination rate of $10 - 15\%$, while with the basic implementation of CRIU, approximately 30% of the selected target instances were of types with an interruption rate exceeding 15%. In the worst case of CRIU default implementation, this ratio can be greater than 20%, which can affect reliability.

Figure 7b presents the effectiveness of the proposed system with respect to the cost savings by using spot instances. The cost savings indicate the ratio of spot instance price to the on-demand instance price for each selected instance type. Similarly to the improvement of the interruption ratio, the expansion of migration candidate nodes using the proposed system is expected to choose spot instances with higher cost savings. In the median case, the proposed execution path tracking achieves 85% savings compared to on-demand prices, while the CRIU implementation achieves 69% savings. In the lowest cost savings scenario, the proposed methods achieve 61% savings, while the CRIU implementation achieves 53% savings, showing a difference of approximately 8%.

In summary, we answered all three research questions. For **RQ-1**, the execution path tracking method could achieve a precision of 1.0 and had almost perfect recall. Compared to the default CRIU implementation, this method has over $5\times$ better recall for diverse workloads. For **RQ-2**, execution path tracking significantly lowers operation overhead compared to the text-segment full scan method, and it took less than 15 seconds at most for the workload analysis. Regarding **RQ-3**, by applying the proposed method in a cloud-based spot-instance environment, the cost savings can be expected to improve by 16% while increasing the reliability.

VI. Related Work

Barbalace et al. [32] 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 [33]. 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 of machines with distinct CPU features.

Few studies have applied the live migration process to enhance computing resource usage efficiency. Juric et al. [34] 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 [4], [35]. Their objective is to minimize resource waste when user sessions are inactive. Cunha et al. [36] 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 mentioned work, the proposed migratable instance detection algorithms can increase target instances, significantly boosting migration efficiency.

VII. Conclusion and Future Work

Live migration of applications on various cloud instances can improve efficiency, as the demand for computing resources for execution varies between diverse scenarios. In heterogeneous cloud instance environments comprising distinct CPU features, guaranteeing compatibility between the source and destination instances is crucial for flawless operation. A simple heuristic can consider all the CPU features of the source and target instances, regardless of the application. However, the precise detection of relevant CPU features that a workload really uses can widen the pool of resources for migration targets. To achieve this goal, we proposed a workload-aware, live-migratable instance detection system. With a thorough analysis of the instructions for a workload, the proposed system can identify the mandatory features to execute a workload flawlessly after migration. The results of the experiment reveal that the proposed system increases the recall of migratable instance detection by $5\times$ compared to the naive approach adopted by CRIU. To show the practicality of the proposed system, we applied it to a cloud spot-instance environment where interruption events can result in application live migration. Using the proposed algorithm makes the migration target instance pool larger, and it enhances the cost savings by 16% with increased reliability.

The proposed execution path tracking module needs to be improved to track untraceable cases of calling functions using a register value to improve the proposed system further. The current implementation supports only the compiled languages, such as C and C++, and we are working to enhance the system to support scripting languages, such as Python.

Acknowledgment

This work is supported by the National Research Foundation (NRF) Grant funded by the Korean Government (NRF-2020R1A2C1102544), AWS Cloud Credits for Research program, and the SW Star Lab (RS-2022-00144309) of IITP.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] GCC. Function multiversioning, 2012.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Intel. Intel® 64 and ia-32 architectures software developer's manual. vol2. 3.3., 2023.

[18] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99(2013):57, 2013.

[19] Chao Zhang, Lei Duan, Tao Wei, and Wei Zou. Secgot: Secure global offset tables in elf executables. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 995–998. Atlantis Press, 2013/03.

[20] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 243–255, New York, NY, USA, 2015. Association for Computing Machinery.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, Boston, MA, August 2022. USENIX Association.

[30] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 266–282, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Kyunghwan Kim, Subin Park, Jaeil Hwang, Hyeonyoung Lee, Seokhyeon Kang, and Kyungyong Lee. Public spot instance dataset archive service. In *Companion Proceedings of the ACM Web Conference 2023*, WWW '23 Companion, page 69–72, New York, NY, USA, 2023. Association for Computing Machinery.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.