Long-Term Preservation of a Software Execution State

Rafael Gieschke
University of Freiburg
Germany
rafael.gieschke@rz.uni-freiburg.de
0000-0002-2778-4218

Klaus Rechert
University of Applied Sciences Kehl
Germany
rechert@hs-kehl.de
0000-0002-2454-4374

Euan Cochrane
Yale University Library
U.S.
euan.cochrane@yale.edu
0000-0001-9772-9743

Abstract – Software is a very complex product, offering an endless number of different states and appearances. To foster academic discussion about software-based cultural and scientific phenomena like computer games, digital art, or scientific computational models, it is necessary to be able to reference specific moments of running software. In this article, we discuss the possibilities to "freeze" software while being executed and describe constraints for the long-term preservation of these snapshots.

Keywords – Emulation, Program Snapshots

Conference Topics – Immersive Information

I. Introduction

Citation is an important tool for scientific research. With citations, one can point to prior or related work, for comparison and scientific or socio-cultural discussion. Hence, access to cited work is an essential service of libraries and archives. Long-term digital preservation is the tool to ensure the availability of (cited) digital works for future access.

With the emergence of more and more complex born-digital works, such as software-based art, computer games, and all kinds of interactive digital artifacts, and software in general, citation of works as well as access to cited works became a significant challenge. Citation of software has been integrated into scholarly practice, firstly due to the increased importance of software as a tool for research, e.g., data processing and data modeling. Especially since scientific software is usually made for a specific purpose, requires dedicated resources, and is an indispensable part of the research process and, thus, their authors deserve attribution. Furthermore, software setups perform extensive and complex operations. With the requirement to reproduce scientific results, availability and ideally re-executability of software is curial.

With emulation, there is a rather generic approach to keep software artifacts usable [1]. Providing a usable (or executable) reference to access and use software artifacts is currently ongoing work, e.g., through the Emulation as a Service Infrastructure (EaaSI) program of work [2]. A working and usable infrastructure to (re-)execute or "re-perform" a preserved digital object is necessary but not always sufficient to support research activities. Software or digital objects in general are not only complex in technical terms and, thus, need necessary technical infrastructure to be rendered but also complex in use and operation and manifold in their options, appearance, and how they are perceived by users. Hence, in many cases the description of the software and its execution dependencies is not sufficient to describe specific aspects of the software's performance. The basic assumption of software functionality (and computers) to process data input and (re-)produce an output does not cover intermediate steps or user-machine and machine-machine interactions during execution. These, however, do not only bear important aspects of how a particular result has been achieved but may constitute important information or facts themselves. For instance, a

iPRES 2023: The 19th International Conference on Digital Preservation, Champaign-Urbana, IL, US.

https://hdl.handle.net/2142/121093

computer game or a software-based digital artwork can produce an almost endless combination of states (e.g., the state after a character in the game has been positioned in a particular way after responding to other game elements). Not all of them are relevant, but some are. Picking these for scholarly discussion requires a way for future readers to explore these states to contribute to the scientific discourse.

In general, one can imagine different options providing future scholars access to these states. The first and technically most viable as well as possibly the most cost-effective option is to create (static) documentation, e.g., through screenshots or video captures [3]. However, this method limits future research options, e.g., it will be difficult for future researchers to explore and inspect the context as well as possible follow-up states.

If additionally, the software and its execution dependencies are preserved and accessible, the documentation can be used to perform necessary steps manually to reproduce a certain state or result. In most cases, however, a manual “replay” of documented steps can be difficult and time consuming. The documentation may not always be sufficient to guide future users successfully through the process, since users may lack skills and/or implicit operational knowledge. Especially computer games require some training to perform certain actions successfully. Furthermore, it is still not assured (especially in the case of a manual re-play) that the state reached is identical to the one described in literature. A potential technical solution to this problem is capturing user-interactions and support automated deterministic replay. Even though this approach shows promising results, there are still a lot of difficulties to be solved for generic use-cases, due to non-deterministic events or behavior of the system, especially with highly dynamic interactions [4].

A further option is the possibility to "freeze” the execution of software at any chosen state as an execution snapshot. Ideally, such snapshots could be archived, shared with others, and restored at any time to continue the execution of the software. While creating snapshots of running virtual machines (VMs) is a common feature of most virtualization and many emulation systems, reactivation in (slightly) differing technical setups as well as long-term usability of snapshots is an open issue. In this article, we investigate this problem and describe a way to implement long-term available “frozen” snapshots of running software.

1. Related Work

With the growing importance of software as a cultural and scientific product, the necessity of citing software was recognized [5]. Citation practice ranges from informal mention to detailed version information but are being formalized further. Citation of software implies (or should imply) the availability of software.

Even though many initiatives have started archiving software, software preservation in general is still an open issue because of the complexity and size of the task and sometimes due to legal obstacles. Preservation of source code [6] is to a certain degree the most systematic and successful public approach so far. However, available source code does only cover a limited field (open-source software) and, more importantly, there is a long-term usability problem [7], as source code cannot be trivially reused without further technical steps (i.e., compilation into an executable binary form). Furthermore, for more complex scenarios, different software products need to be contextualized (setup and configured) within a defined execution environment (e.g., using an emulator) and, especially in context of computational research, brought together with data. Container technology solved some of the dependency, setup and configuration problems [8] but seldomly reduces the operational complexity (e.g., the number of settings, runtime variables) of non-trivial setups.

For smaller, controlled subsets, there are tailored solutions, not only to reproduce a computational result but also to transparently document all in-between steps in an executable form, e.g., Jupyter notebooks. But even for these rather constrained niches, long-term access and reproduction is quite difficult [9], [10].

Another alternative to deterministically restore a specific application state without reproducing interactive user (GUI-)events (e.g., [11]), which are prone to non-deterministic behavior of GUI applications, are so called “record and replay” tools (e.g., [12], [13]). In contrast to tools recording GUI-events (e.g., macro recorders), these tools record the interaction of an application with the underlying operating system and, thus, can deal with potential

non-determinism (e.g., clock/time functions, random number generators). These tools, however, are usually quite intrusive (e.g., they depend on special CPU features or only work for special cases like applications with limited interactivity) and slow down the execution of the recorded application significantly.

2. Snapshots of Running Application Processes

Presumably, the main motivation to access preserved (and cited) software setups beside verification of published results could be exploring the relevant software environment further, e.g., by changing intermediate inputs in a case of interactive computational science or to follow different paths of a computer game. But there are many other different use-cases for preserving a specific state of a running software setup. This section will discuss the concepts to generate software snapshots.

A "snapshot" is a saved (serialized) state of a running process or system. Any information needed to restore the process or system and to continue its execution is contained within the snapshot. This operation requires some technical support of the surrounding environment, i.e., the software that is creating the snapshot. Since we do not want to constrain snapshots to a single architecture, computer platform, or operating system, i.e., we want to be able to cover applications from different technical epochs, we chose to take a snapshot of the whole system the application of interest is running in (guest system). We, thus, assume that the guest system is running on either virtualized or emulated hardware (i.e., in an emulator or hypervisor).

In principle, there are different ways to create a snapshot of a running system. The first one is taking advantage of the capabilities of the hypervisor or emulator. Some emulators (e.g., QEMU, v86) or VM hypervisors (e.g., VMware Workstation/ESXi, VirtualBox) offer built-in functionality to create snapshots. The emulator or hypervisor can pause the execution of a guest and save all states of the hardware (i.e., the virtual CPU and any other virtual hardware devices). The snapshots created this way are called *virtual machine snapshots*. Saving the state of the whole computer system to disk can also be done by the (guest) operating system itself without the help of a hypervisor, a feature typically called "hibernation" or "suspend to disk". However, this option is only available for a few emulators/hypervisors (and guest operating systems in case of hibernation).

But having a working option to create virtual machine snapshots (e.g., QEMU), a further option to consider could be to run emulators not capable of creating snapshots themselves (e.g., SheepShaver, VICE) within a capable hypervisor (or emulator) and use it (the "outer" emulator) to create snapshots (of the "inner" emulator). Most likely, the generated snapshots depend on the software (emulator or hypervisor) they were created on. Firstly, virtual machine snapshots are saved (serialized) in a format specific to the emulator/hypervisor software used. Secondly, while it would be possible for other emulators to re-implement support for an existing virtual machine snapshot format, this seems unlikely as the virtual machine snapshots including their configured operating system require the exact same (virtual) hardware (i.e., not only the same CPU but also identical other virtual hardware devices) it has been created on to be restored successfully [14]. Even the forward-compatibility of virtual machine snapshots to later versions of the same emulator/hypervisor will likely decrease over time as their main usage is for live migration and short-term snapshots.[1] Hence, virtual machine snapshots must be carefully curated and maintained together with the necessary software (i.e., the emulator or hypervisor) for restoring, which will lead to significant maintenance overhead over time.

An alternative option to storing the complete hardware state of a virtual machine is to create *process snapshots* (also known as application snapshots) which do not rely on a hypervisor but on functionality offered by the operating system. Process snapshots save the state of a specific running application, comparable to the way the

[1] We have found that a snapshot taken using hibernation by the guest operating system might even fail to restore on different builds of the very same emulator source code, see https://gitlab.com/emulation-as-a-service/emulators/qemu-eaas/-/blob/v2.5/dsdt.patch. This finding might be extendable to virtual machine snapshots taken by the emulator/hypervisor but does require further research.

operating system saves the process's state when executing a context switch.

While virtual machine snapshots save the state of the whole platform (i.e., the CPU and any devices, e.g., the IBM PC platform), process snapshots only save the state of (parts of) the CPU (i.e., only the parts of the hardware defined by the instruction set architecture (ISA), e.g., the x86-64 architecture). For other resources (the equivalent of devices), they make use of the abstraction the operating system provides and are therefore less hardware dependent. Consequently, however, they have a rather strong operating-system (i.e., kernel Application Binary Interface "ABI") dependency as well as a dependency on any open files, network connections, or similar resources originally being in use by the process at the time of creating the snapshot. For our purposes, we chose to create a process snapshot of the emulator process[2], and thus, implicitly the state of the guest system.

Taking a snapshot of a running process usually requires special support from the operating system, because some parts of the process's states are not visible to the process itself but only available to the operating system. The Linux kernel was extended to provide previously missing APIs to create process snapshots by the *Checkpoint/Restore In Userspace* (CRIU) project[3], which also provides a user-space helper utilizing the kernel's API to create, serialize (i.e., save as regular files), and restore process snapshots[4].

CRIU works best when it is used to snapshot isolated applications running in their own Linux container[5] using a container runtime[6] and is, in fact, already well integrated into contemporary container runtimes[7]. As we already developed a framework to package (untrusted and potentially insecure) emulators including supporting infrastructure as self-contained container images and let them run in an isolated Linux container in the EaaS emulation framework [15], we used the same approach for CRIU[8]. Taking this approach, preserving emulators (and snapshots) becomes a special case of preserving containers [16].

3. Long-Term Access to Process Snapshots

Solving portability issues of the created snapshots solves only the smaller part of the problem. The second problem set is to restore or reactivate a snapshot using future computer systems. By choosing process snapshots, we have reduced hardware dependencies, which are typically abstracted by the operating system. For instance, for a (user-space) application to be run on a Windows or Linux system, the hardware configuration of the computer does typically not matter to the application. Instead, an application uses well defined interfaces of the operating system to interact with graphics, sound, or network hardware. These interfaces are stable and do not change, e.g., if a hardware component is replaced. Hence, for process snapshots, the remaining (future) dependencies are the operating system application binary interface (ABI) and the CPU instruction set architecture (ISA;

[2] We are only using emulators here. There would be no principal problem in creating a process snapshot of a hypervisor that uses the Linux kernel's KVM ABI (e.g., QEMU) as these hypervisors are normal processes. However, this functionality has only niche use cases and is not supported by process snapshot frameworks, see https://github.com/checkpoint-restore/criu/issues/229.

[3] Checkpoint/Restore In Userspace, see https://criu.org/.

[4] The kernel patches created by the CRIU project were accepted and included in the upstream Linux kernel, so that the CRIU user-space helper can be used together with any stock Linux kernel, reducing maintenance burden.

[5] See, e.g., https://www.redhat.com/en/topics/containers/whats-a-linux-container.

[6] Standardized in the OCI Runtime Specification (https://github.com/opencontainers/runtime-spec) and implemented in, e.g., Docker, runc, or crun with the help of functionality provided by the Linux kernel.

[7] See, e.g., https://github.com/opencontainers/runc/blob/main/man/runc-checkpoint.8.md.

[8] The images may have to be slightly altered (e.g., checked in an automated process) as CRIU can place subtle restrictions on the kind of executables it is able to restore properly, see https://github.com/checkpoint-restore/criu/issues/1507.

e.g., x86-64), which should make these snapshots portable between contemporary systems but also to future systems.

1. *Restoring the Execution Context*

When CRIU restores a snapshot, it interacts with the Linux kernel to restore the exact same state the process was in when the snapshot was created. The "exact same state" reproduced by CRIU includes CPU registers and memory used by the application as well as any resources opened by the application, e.g., files or network sockets. These resources must remain available in the very same state (e.g., the same file content) and must be saved, managed, and restored independently from the CRIU snapshot. We have already accomplished this by using container images, for which a derivative image can be created when snapshotting the application. For network connections (e.g., connected TCP sockets), CRIU's approach is usually more brittle as they depend on the uncontrollable outside world (e.g., the remote end of a TCP socket will probably be gone when the snapshot is restored). In the EaaS emulation framework, we have already solved this problem by providing network access only via a virtualized and isolated network.

2. *Identifying Remaining Hardware Dependencies*

The described approach makes sure that any applications can continue to run after being restored. However, CRIU's intended usage scenarios[9] are focused on short-term usage of the created snapshots on homogenous machines (i.e., using the same CPU model or, at least, the same CPU generation from the same CPU vendor with the same architectural features, similar other installed hardware devices, and similar Linux kernel versions). In other words, CRIU's process snapshots may still depend on the original CPU model. This can be a surprising property as applications can break in unexpected and subtle ways under different CPU models, even more so when the CPU is changed during the application's execution. Normal applications do not expect that the CPU's (visible) features change during execution[10] and, as the application, deliberately, has no way to notice that it is being snapshotted and restored, it even has no chance to react to the changed CPU at all.

On the x86(-64) architecture, the CPU vendor and model, its supported extensions to the original ISA, and other features of the CPU are exposed via the CPUID machine instruction. The CPUID instruction is unprivileged and, thus, can be executed by user-space applications. It is typically, transparently, included by the compiler into the executable binary[11] for features like function multiversioning[12] or queried using dedicated libraries[13] by the application itself. All these features have in common that they, at the start of the application, select one of several machine code versions of the same function most suitable (i.e., optimized) for the current CPU model. This is a problem if the target CPU has less features (e.g., no AVX2) than the original CPU. After resuming the snapshot, the application will still execute the code path requiring the original CPU features (e.g., AVX2) and execution on the target CPU will fail, generating a SIGILL signal (on Linux/POSIX) and terminating the process[14].

A possible approach could be to require the target CPU to always support more features than the original CPU, e.g., by requiring it to support all CPU extensions available at a time. We found, though, that this approach does not work as, apart from

[9] https://criu.org/Usage_scenarios

[10] For their recent processors with heterogeneous CPU core configurations, Intel takes great pain in ensuring that all cores expose exactly the same process-visible features, see, e.g., https://www.tomshardware.com/news/intel-nukes-alder-lake-avx-512-now-fuses-it-off-in-silicon.

[11] See, e.g., gcc and its libgcc: https://github.com/gcc-mirror/gcc/blob/59a72acbccf4c81a04b4d09760fc8b16992de106/gcc/common/config/i386/cpuinfo.h#L975.

[12] https://gcc.gnu.org/onlinedocs/gcc/Function-Multiversioning.html

[13] See, e.g., libcpuid: https://github.com/anrieff/libcpuid.

[14] While it would be imageable to catch this signal and emulate just the missing (e.g., AVX2) instructions, in practice, this approach is unfeasible as other (existing) (SSE) instructions modify the state of the (extended) registers used by AVX2 as well while not producing a catchable SIGILL instruction. It is, thus, not possible to selectively emulate these instructions but all instructions (including the existing ones) would have to be emulated.

being expensive by requiring the latest CPUs, CPU features are not only added by vendors but sometimes also removed again in later CPU model generations[15]. A ”best“ CPU including every feature ever introduced may, thus, not always exist.

More importantly, though, we found that, e.g., a snapshot created on a CPU not supporting AVX2 will crash on a CPU supporting AVX2, i.e., a CPU with strictly more features. Debugging showed that this is due to the XSAVE instruction, which writes the CPU register's state to memory at an application-provided location. The (byte) size the register state will need in memory can be queried by the application (or a supporting library) using the CPUID instruction, but this, again, is typically only done at application startup and cached for later use. If, in our case, the snapshot is now resumed on a target CPU requiring more space for its register state (i.e., by having the larger AVX2 registers), not enough space will have (unknowingly) been reserved by the application when calling the XSAVE instruction and the CPU will silently overwrite parts of memory, e.g., in the application's stack, leading to the application's crash.

A possible workaround is to disable the problematic features (e.g., AVX2) on the target CPU with the operating system's help[16]. This workaround was tested but proved not to be successful as a number of applications, including the widely used GnuTLS library[17], check incorrectly for the availability of CPU features, subsequently still try to use the disabled features and crash. This results in crashes of even very basic dependent applications like ”apt-get”.

Thus, the most feasible way is to directly restrict the CPUID instruction to report only desirable features as available and report a buffer size for the XSAVE instruction that is large enough for any target CPU's register state. At the same time, this is advantageous as it allows to restrict the used CPU features to a sensible set of features supported by not only the latest CPUs but a large number of (cheaply available) CPUs from different vendors (and most x86-64 emulators). Such “common denominators” of features are already standardized as micro-architecture levels (e.g., x86-64-v2) in the ELF x86-64 psABI[18] and are recently starting to be used to define minimum system requirements of Linux distributions.

CPUID virtualization

Manipulating the CPUID instruction, though, proves to be problematic on the x86-64 architecture, a property than can be attributed to the fact that the x86-64 architecture (without extensions) does not conform to the virtualization requirements introduced by Popek and Goldberg [17]. The CPUID instruction can be seen as a behavior sensitive instruction that is not privileged, i.e., can be executed directly by user-space application without a chance for the operating system's kernel to interfere.

An extension found on most Intel processors[19] remediates this problem by allowing to turn the CPUID instruction into a privileged instruction, and, thus, trapping it in user space.[20] This feature is used

[15] Again, see the removal of AVX-512 from Intel's recent processors with heterogeneous CPU core configurations, https://www.tomshardware.com/news/intel-nukes-alder-lake-avx-512-now-fuses-it-off-in-silicon.

[16] Using the Linux kernel's ”clearcpuid” and ”noxsave” command-line options, which do not directly interfere with the result of the CPUID instruction for user-space applications but only disable CPU features using the CR4 control register.

[17] See https://gitlab.com/gnutls/gnutls/-/issues/1282. The underlying problem is that such problems do not get much real-world test coverage as users do not typically want to restrict their CPU's features, i.e., never run applications under such kernel configurations.

[18] Processor-specific application binary interface, https://gitlab.com/x86-psABIs/x86-64-ABI.

[19] Exposed by the Linux kernel via the arch_prctl(ARCH_SET_CPUID, ...) system call, see https://man7.org/linux/man-pages/man2/arch_prctl.2.html.

[20] Both Intel's VT-x and AMD's AMD-V virtualization extensions for the x86-64 architecture do allow for hypervisors to trap CPUID instructions. An alternative approach would, thus, be to have the application run in its own virtual machine (potentially inside yet another virtual machine provided by a

by the libvirtcpuid project[21], which, independently from our work, researched the presented problem of a snapshot created on a CPU with less features (e.g., no AVX2) crashing when being resumed on a CPU with more features (e.g., AVX2). Their intended use case, however, focuses on live migration of applications and differs from our long-term preservation use case. Additionally, they rely on the described Intel extension, which is neither found on AMD x86-64 CPUs nor in emulators and often not exposed by cloud providers, reducing its usefulness for our application significantly.

We, thus, modified libvirtcpuid's approach slightly: while the original libvirtcpuid relies on the described CPU extension to trap CPUID instructions, we modify the application's executable (ELF) binary files in advance to replace every CPUID instruction with an RDMSR instruction[22]. RDMSR instructions, in turn, are always privileged on the x86-64 architecture and, thus, will trap with a signal to user-space that can be processed by libvirtcpuid in its usual way[23]. This approach is not guaranteed to work as applications may dynamically generate just-in-time (JIT) code including CPUID instructions. Only being generated at runtime, these instructions would not be processed by our tool and still leak unmodified CPUID information to the application. However, this is very unlikely as, as described above, code using the CPUID is typically statically generated by the compiler or placed in dedicated libraries. Other potential problems, e.g., applications checking for CPU features by trying to directly use them without checking for their availability first[24], are the same as for the upstream libvirtcpuid and rare in practice.

Other sources of non-determinism

As described above, the restored application also depends on outside resources. Obvious ones like files (either regular files, UNIX domain sockets, or pipes) are already handled by our emulation framework. Non-obvious ones include time functions[25], which naturally depend on the time in the outside world, either measured as time since the system was booted (CLOCK_MONOTONIC) or as real (i.e., wall-clock) time (CLOCK_REALTIME). The former (CLOCK_MONOTONIC) can be virtualized by the Linux kernel using time namespaces[26], which are already supported and handled by CRIU. As they are typically used via the C standard library, the latter (CLOCK_REALTIME) can potentially be modified in user-space[27] to be derived from CLOCK_MONOTONIC with a constant (user-configurable) offset.

However, an application could also use the RDTSC instruction to directly read the processor's

cloud provider) restricted to, e.g., the x86-64-v2 micro-architecture level. This approach was not pursued as more virtualization levels will most probably degrade performance and (nested) virtualization is not universally available at cloud providers or comes with extra costs, see, e.g., https://ignite.readthedocs.io/en/stable/cloudprovider/.

[21] https://github.com/twosigma/libvirtcpuid

[22] Introducing the ELFant tool, a powerful but friendly shell script that tramples over your ELF files, see https://github.com/emulation-as-a-service/libvirtcpuid.

[23] A conceivable alternative approach of replacing CPUID instructions with a call to a library function emulating and manipulating the CPUID instruction directly is not feasible as the CPUID instruction is encoded in only 2 bytes, which is not sufficient space for any jump/call instruction. In contrast, the RDMSR instruction is encoded in 2 bytes as well, and can, thus, directly replace the CPUID instruction in the binary file without any further modifications. Yet another conceivable alternative approach of replacing library functions (e.g., libcpuid) utilizing the CPUID instruction is not feasible as they might not easily be recognized anymore when linked into the application in binary form, the variety of such libraries is too diverse, and some applications (e.g., GnuTLS) do not employ such libraries at all but directly use the CPUID instruction for similar purposes, leading to a very brittle application-specific manual patching approach.

[24] In this case, if the CPU feature is available on the original CPU, the application will recognize it as usable but will crash as soon as trying to use it on the target CPU after the snapshot is resumed.

[25] Exposed via the clock_gettime() system call on Linux, see https://man7.org/linux/man-pages/man3/clock_gettime.3.html.

[26] https://man7.org/linux/man-pages/man7/time_namespaces.7.html

[27] For instance, using the LD_PRELOAD mechanism, see https://man7.org/linux/man-pages/man8/ld.so.8.html.

time-stamp counter. Differently from the CPUID instruction, the RDTSC instruction can be disabled and trapped in user-space[28]. As the application might not expect RDTSC to not work, further modifications to the application might be necessary. In contrast to the CPUID problem, however, this problem is immediately visible already when starting the application on the original CPU and can, thus, be handled more easily instead of manifesting itself in an unfixable way later long after the snapshot has been created.

A final class of resources that must be dealt with are resources available on the original system that are being masked by the container runtime/configuration but are not available at all on the new system. As they are not available at all, CRIU cannot mask them like on the original system and, thus, might fail to restore the snapshot.[29] Here, a simple workaround is to not mask the resources in the first place.[30]

5. *Emulation in Emulation*

To restore an application state, the snapshotted emulator process containing the running application must be restored using a suitable technical environment. In the case of a future reactivation, the required technical environment for restoring will be an appropriate emulator, e.g., a suitable x86-64 emulator satisfying the CPU dependencies (e.g., the x86-64-v2 micro-architecture level) and a container runtime including an appropriate Linux kernel and pre-configured tools to restore the snapshot (e.g., CRIU). This runtime then will be able to restore the snapshotted process (emulator running a guest system). This setup, however, will lead to an emulation-in-emulation (stacked emulation) access context.

Using emulation-in-emulation as an emulation-based digital preservation strategy is not an ideal solution in general. While the idea is simple and appealing, i.e., today's emulator setups containing obsolete systems and running, e.g., on a contemporary Windows 11 system, can be preserved and kept available through future emulators by simply focusing on today's Windows 11 system and so on. However, with technical epoch and thus, every new level in this emulator stack, a technical and conceptual mapping between contemporary computer systems and the last generation (latest emulators) must be made.

These mappings usually require technical compromises, especially but not only, for interactive usage, because future concepts have changed significantly. For instance, by moving toward gesture inputs, mapping modern touchscreen gestures to 3-button mouse events is necessary. Clever emulator developers will find a user-friendly and usable solution for their technical environment and context. However, with every additional layer (and mapping), it will be harder to operate the original system (e.g., the 3-button mouse, through an emulated touchscreen using a future VR setup) and, eventually, some states or concepts of the original system will become inaccessible throughout the different layers. Therefore, we usually argue to "migrate" the old (guest) systems to new emulators, such that any new generation of computer systems with new interaction paradigms (e.g., virtual reality) adapt these directly to the old concepts.

For this special case (snapshots of running processes), however, the emulation-in-emulation scenario is necessary and justifiable. Its necessity results from the design choices of taking process snapshots. In the case of virtual machine snapshots, restoring the saved machine state on future emulators is possible in theory. However, in practice it will be quite difficult since the target machine must match exactly the hardware of the snapshotted virtual machine. Even the transition (live migration) between two contemporary virtual machines, both running within the same emulator (in this case, two

[28] Exposed as prctl(PR_SET_TSC, …) by the Linux kernel, see https://man7.org/linux/man-pages/man2/prctl.2.html.

[29] For instance, the Linux kernel only provides /proc/asound/ (masked as empty directory inside containers) if the host system includes a sound card. This is typically true for desktop computers but untrue for server or cloud computers. A snapshot created on a desktop computer may, thus, fail to resume on a server or cloud computer.

[30] This can pose security problems but, e.g., in the sound card scenario, it might be acceptable to not mask the sound card when being run on a desktop computer as the desktop computer is typically operated by only one user, who is already able to manipulate and interfere with the sound card in host system anyway.

QEMU-based VMs) turned out to be rather difficult [18].

For the conceptual idea of saving and restoring an application state, the concept of emulation-in-emulation is appropriate. Not only is the depth of the emulation stack limited to the maximum of one extra layer and, thus, the mapping problem between contemporary systems and old systems is addressable, but this also is a desired setting since the future user is able to observe the exact state, including all features and limitations of the access platform (emulator) the creator of the snaphot has experienced. Even if the restored snapshot offers limited usablity (compared to running the application using future emulation-based access platforms), it offers a stable and reproducible reference point.

4. Conclusion and Future Work

In this article, we have presented a technical and conceptual analysis on the preservation of software execution states (so called snapshots). The concept of preserving snapshots will not only contribute a further facet for software citation, but it will also contribute to an increased usability of emulation setups by simplifying the preparation of ready to use software setups. Users can be presented with a configured and running application in a usable state, without the hurdle of operating an old computer system, e.g., staring an application, finding the necessary files, etc. For some (future) software setups, such an approach might be the only viable solution. Software becomes a boundless product which is difficult to capture and to "own" since it is not shipped anymore on media. Modern software offers dynamic installation processes with multiple options for extension, in-app purchases, etc., relying not only on individual decisions but also the publisher's infrastructure and especially support of installing outdated software packages.

In addition to citation use cases, the functionality required to fulfill these can be useful for other long-term access purposes. For example, when making digital artifacts available via emulation that rely on slow or complex software environments, it can be valuable from a user-experience perspective to be able to immediately restore the system or network state to a point after all the software components involved have completely loaded. The state-saving functionality described in this paper is currently in use at Yale University Library as part of their efforts to retain access to web sites including the Ross Archive site[31] and the Historical Register Online site[32]. Access to both sites is provided via emulation of hardware supporting the underlying web server software, database software, and a contemporaneous web browser. Each of these has to be loaded before the user can access the site in emulation and the process can take minutes. By pre-loading them, saving the state in a process snapshot, and loading the state at point of access, this saves the user a great deal of time and significantly improves the user experience for them.

While there is a proof-of-concept implementation available as well as working real-world examples as part of the EaaSI project, the focus of this work was to improve our understanding of potential technical hurdles restoring snapshots in the future. x86-64-based systems are still the most important platform running in-production framework. Hence, our analysis was focused on x86-64 process snapshots. Future work will widen the scope to other relevant platforms (e.g., ARM64).

Re-storing snapshots does not only pose technical challenges but also administrative ones. The runtime environment must be archived and maintained, as well as any other runtime dependency. By encapsulating emulators with their runtime dependencies within containers, preserving containers (and their runtime), the preservation of snapshots is just a special case of the existing EaaS container preservation workflow.

An important limitation of our experimental setup is the time lag between a snapshot request and the snapshot executions. This lag is currently somewhere around 2-5 seconds. Additionally, serializing a snapshot takes time – linear to the total memory of the process used. Hence, currently our setup is not yet suitable for highly dynamic software setups, e.g., computer games, where states are changing very quickly, and not suitable to create a series of fine-grained snapshots. However, there are promising developments to improve snapshot

[31] https://rossarchive.library.yale.edu/

[32] https://yalehistoricalregister.library.yale.edu/

performance and to support fine-grained, high frequency incremental snapshots [19].

5. REFERENCES

[1] D. S. Rosenthal, "Emulation & virtualization as preservation strategies," 2015.

[2] E. Cochrane, K. Rechert, J. Oberhauser, S. Anderson, C. Fox, and E. Gates, "Useable Software Forever," *IPres 2022 Glasg. 12—16 Sept. 2022 Www Ipres2022 Scot*.

[3] J. P. McDonough *et al.*, "Preserving virtual worlds final report," 2010.

[4] J. Oberhauser, R. Gieschke, and K. Rechert, "Automation is Documentation: Functional Documentation of Human-Machine Interaction for Future Software Reuse," *Int. J. Digit. Curation*, vol. 17, no. 1, Art. no. 1, Sep. 2022, doi: 10.2218/ijdc.v17i1.836.

[5] A. M. Smith, D. S. Katz, and K. E. Niemeyer, "Software citation principles," *PeerJ Comput. Sci.*, vol. 2, p. e86, 2016.

[6] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli, "Referencing source code artifacts: a separate concern in software citation," *Comput. Sci. Eng.*, vol. 22, no. 2, pp. 33–43, 2019.

[7] K. Rechert, J. Oberhauser, and R. Gieschke, "How Long Can We Build It? Ensuring Usability of a Scientific Code Base," *Int. J. Digit. Curation*, vol. 16, no. 1, p. 11, 2021, doi: 10.2218/ijdc.v16i1.770.

[8] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, 2015.

[9] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, 2019, pp. 507–517.

[10] J. Wang, T. Kuo, L. Li, and A. Zeller, "Assessing and restoring reproducibility of Jupyter notebooks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 138–149.

[11] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 1535–1544.

[12] O. S. Navarro Leija *et al.*, "Reproducible containers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 167–182.

[13] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering Record and Replay for Deployability.," in *USENIX Annual Technical Conference*, 2017, pp. 377–389.

[14] K. Rechert, O. Stobbe, O. Zharkow, R. Gieschke, and D. Wehrle, "CiTAR - Preserving Software-based Research," *Int. J. Digit. Curation*, vol. 15, no. 1, Art. no. 1, 2020, doi: 10.2218/ijdc.v15i1.716.

[15] R. Gieschke and K. Rechert, "A Generic Emulator Interface for Digital Preservation," in *iPres 2022 Glasgow 12—16 September 2022 www. ipres2022. scot*, 2022.

[16] K. Rechert, T. Liebetraut, D. Wehrle, and E. Cochrane, "Preserving containers–requirements and a todo-list," in *Digital Libraries: Knowledge, Information, and Data in an Open Access Society: 18th International Conference on Asia-Pacific Digital Libraries, ICADL 2016, Tsukuba, Japan, December 7–9, 2016, Proceedings 18*, 2016, pp. 225–230.

[17] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[18] J. Wei, L. K. Yan, and M. A. Hakim, "Mose: Live migration based on-the-fly software emulation," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 221–230.

[19] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory CRIU for docker containers," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 53–65.