Portability of a CRIU-based Java process checkpoint

By Ashutosh Mehra, Red Hat
asmehra@redhat.com
Feb 3, 2022

There are multiple ongoing efforts in using CRIU, a tool to checkpoint and restore a process on Linux, in the context of a JVM. They aim at leveraging CRIU to create a checkpoint of the process after the start up phase of a JVM-based application has completed. This checkpoint can then be used to restore the process for subsequent invocations of the application, thus bypassing all the time-consuming startup work that JVM has to do on every invocation.

Restoring the process from the checkpoint can, in principle, be done on any system where the checkpoint image is available. So we did some experiments to test out the “portability” of the checkpoints of the Java processes taken by CRIU.

Setup

OpenJDK has a project CRaC which aims at using CRIU to create checkpoints. We used this project to create a checkpoint of the Java process using Hotspot as the JVM. The Java program was a simple HelloWorld application:

```
public class HelloWorld {
     public static void main(String args[]) throws Exception {
          System.out.println("Before checkpoint");
          jdk.crac.Core.checkpointRestore();
          System.out.println("After Restore");
     }
}
```

The call to `jdk.crac.Core.checkpointRestore()` would result in invoking CRIU to create the checkpoint of the Java process.

The command to run this program and create the checkpoint is:

```
# java -XX:CRaCCheckpointTo=cr HelloWorld
```

Above command would terminate the java process after creating the checkpoint in the directory name `cr`.

Portability

We can consider portability of the checkpoint with respect to hardware and operating system of the two systems under consideration - *S* where the checkpoint is taken and *D* where the process is restored. This gives us four configurations:

1. Same hardware, same distro
2. Same hardware, different distro
3. Different hardware, same distro
4. Different hardware, different distro

Out of these four, our main focus in this experiment was on configuration 3 as it allows us to check the portability of the compiled code generated by Hotspot (which is another goal of this exercise). Both configurations 1 & 3 represent the common containerized use cases as the container image would hold the distro constant while allowing the hardware to vary.

1. Same hardware, same distro

In this case *S* and *D* have the same hardware and run the same Linux distribution.
For this setup we created two virtual machines on the same host running Fedora 30.
Creating a checkpoint on *S* and restoring on *D* worked fine, which was expected as there is no change in the process environment.
However, any upgrades on D done between the checkpoint and restore can cause issues when the application resumes after restore.
For instance, the system library updates may change the layout of the functions causing the library function calls to jump to an address which may not be valid in the updated library.

2. Same hardware, different distro

In this case *S* and *D* have the same hardware and run different Linux distributions, or different versions of the same Linux distro.
Here the restore process by CRIU can run into problem due to multiple reasons:

1. Different glibc versions
2. Different location of system libraries
3. Different location of system configuration files
4. Different size of configuration files
5. Different VDSO (a small kernel library mapped into user space with kernel version-specific layout & content)

3. Different hardware, same distro

In this case *S* and *D* have different hardware/cpu features but run the same Linux distribution.

In addition to the problem mentioned in configuration 1 related to system upgrades, here the restore process can also run into problems arising due to differences in underlying hardware features and resources. This can manifest in any software involved in the restore process:

1. **CRIU**
    a. CRIU has an option `--cpu-cap` which can be used to impose checks for compatibility of cpu features between the systems involved in checkpoint and restore. If this option is used then the restore process can fail if CRIU determines incompatibility between the systems.
    b. Even if `--cpu-cap` option is not used, restore by CRIU can still fail if the two systems differ in `xsave` cpu feature. The presence of `xsave` feature at the time of checkpoint causes CRIU to store the FPU state as part of the `xsave` area. During restore from a such a checkpoint image, if the system does not have `xsave` cpu feature, then CRIU, by default, will error out with a message:

    ```
    FPU xsave area present, but host cpu doesn't support it
    ```

    This can be worked around by specifying `--cpu-cap=none` which would allow CRIU to restore FPU state and ignore the extended region of the xsave area

2. **glibc/ld:** Many of the library functions like `memset`, `memcpy`, `memmove` have multiple implementations exploiting the cpu features of the architecture. The decision to select amongst them is taken at runtime based on the cpu features available on the system. On first invocation the dynamic linker/loader resolves the function and patches the GOT (Global Offset Table) with the implementation best suited for the system.
This can pose problems during restore on another system if it does not have a cpu feature that the particular implementation of the library function is exploiting.
Even though CRIU may be able to restore the process, the application would eventually suffer SIGILL (Illegal Instruction) when it attempts to execute the library function.
The same applies for an internal function `dl_runtime_resolve_*` in glibc which has multiple implementations and the selection happens during runtime.
Importantly `dl_runtime_resolve_*` is the function that is responsible for resolving the function calls.

    We tried various mechanisms to work around this problem:
    1. **CPUID faulting -** For this work around, it is worth understanding that criu, glibc and Hotspot use `cpuid` instruction to determine the cpu features.
    There is a shared library, *libcpuoverride*, which attempts to override the cpuid instruction using CPUID faulting mechanism.
    In principle, we can use this library during checkpoint and restore to present a consistent set of cpu features to the application, thereby bypassing any issues that may arive due to cpu features incompatibility.

However, this library is not very well tested with different Linux distributions, and didn't work on the set up we have used for our tests.
Moreover, CPUID faulting is exposed in the userspace through `arch_prctl()` system call using `ARCH_SET_CPUID` setting which does not persist across `execve()` calls.

2. **LD_BIND_NOT** - Another option is to use the environment variable `LD_BIND_NOT` during checkpoint. This would prevent ld from updating the GOT after function resolution, and would cause ld to follow the function resolution process for every library call. However, it can still result in `SIGILL` in case the version of `dl_runtime_resolve_*` selected during checkpoint is not compatible with the system where restore takes place. This is because `dl_runtime_resolve_*` is not controlled by `LD_BIND_NOT`.

3. **GLIBC_TUNABLES** - glibc in recent versions added *tunables* that allow applications to alter the runtime behavior of the library. One of the tunables, *glibc.cpu.hwcaps*, allows the user to enable/disable a cpu feature. When starting the application for checkpoint, we can set this tunable to mask off the cpu features which are not present in the system where the process is to be restored. However this tunable comes with a couple of caveats:
    a. It is present only for i386 and x86_64 architectures
    b. Values recognized by the tunable may differ across glibc versions. Check the sources (`sysdeps/x86/cpu-tunables.c` and `sysdeps/x86/cpu-features.h`) to identify the values recognized by this tunable in the glibc version being used.

       N.B. In our experiments on Fedora 30 systems with glibc 2.29, there was an additional issue - this tunable was not able to control the selection of `dl_runtime_resolve_*` function. Fortunately this has been fixed in the version 2.34.

3. **Hotspot:** Hotspot also uses the cpuid instruction to determine the cpu features of the system at runtime. C1/C2 compiler uses cpu features to determine the instructions to be generated at the code generation stage. Therefore, the compiled code generated by C1/C2 compiler can also cause SIGILL if the instruction generated is not available on the system where the process is being restored.

   Fortunately, Hotspot provides an option to disable the use of a particular cpu feature. For example if -`XX:UseAVX=0` is specified, Hotspot will not use the AVX feature at runtime for code generation. Such options can be used to disable the cpu features when launching the application for checkpointing. See Table 1 below shows Hotspot's non-standard options to control cpu features on intel architecture (n.b. this is not an exhaustive list).

Other issues that can pop-up after restore could be due to differences in the hardware resources between the systems involved in checkpoint-restore.
For example, Hotspot computes the number of cpus/cores, cache line size, page sizes (and possibly other hardware resources/configuration) for internal purposes.
Another aspect that can play a role is the change in the underlying OS from virtualized to bare metal or vice-versa.
More investigation needs to be conducted to understand the impact of these changes on the working of the JVM post restore.

4. **3rd party native libraries:** Any other library that the application is linking to may internally exploit cpu features which can cause problems when the application resumes after restore.

Table 1: Mapping Hotspot's non-standard options to cpu feature for intel architecture (subject to change)

Hotspot Option	Default value	CPU features	
-XX:UseSSE=*n*	2	1	sse
		2	sse sse2
		3	sse sse2 sse3 ssse3 sse4a
		4	sse sse2 sse3 ssse3 sse4a sse4.1 sse4.2
-XX:UseAVX=*n*	3	1	avx
		2	avx avx2
		3	avx avx2 avx512f avx512dq avx512cd avx512bw avx512vl avx512_vpopcntdq vpclmulqdq

			avx512_vaes avx512_vnni avx512_vbmi avx512_vbmi2
-XX:[+-]UseAES	false	aes	
-XX:[+-]UseCLMUL	false	pclmulqdq	
-XX:[+-]UseFMA	false	fma	
-XX:[+-]UseSHA	false	sha	
-XX:[+-]UseCountLeadingZerosInstruction	false	abm	
-XX:[+-]UseBMI1Instructions	false	bmi1	
-XX:[+-]UseBMI2Instructions	false	bmi2	
-XX:[+-]UsePopCountInstruction	false	popcnt	
-XX:[+-]UseFastStosb	false	erms	

4. Different hardware, different distro

This is the hardest case as it combines the difficulties presented by configurations 2 and 3 above.

Summary

1. When it comes to portability of the checkpoints, all the software pieces involved in the process (CRIU, glibc, application, other libraries) need to participate to make the checkpoint portable.
2. CRIU based checkpoints seem to be portable across systems with the same hardware resources and OS.
3. With some effort the checkpoints can be made portable on systems running the same OS but different hardware resources.
4. Compiled code generated by Hotspot is not by default portable as it exploits underlying cpu features. But the (non-standard) options present in Hotspot can be used to control the cpu features to be exploited. By careful use of these options, Hotspot can generate more portable code which can run on a variety of systems differing in cpu features. But it should also be noted that preventing Hotspot from exploiting these features can

potentially degrade the quality of the code generated, thereby impacting the performance of the application.

Observations

1. Hotspot would benefit from some kind of *-XX:+PortableCode* option which would trigger generation of more portable code. For that purpose the minimal set of cpu features that the Hotspot be allowed to exploit needs to be determined. Performance of the resulting code would also play a role in determining this minimal set.
2. A mechanism like CPUID Faulting which can allow all software pieces to have a consistent set of cpu features would go a long way in solving the issues mentioned in configuration 3 where the hardware features may vary between checkpoint and restore. However, limitations of the mechanism currently available to use the CPUID Faulting feature at user-level restricts its usability.