Relocation in JIT

Relocation for Hotspot JVM Jitted Code

Yuantian Ding

3 Category of Relocation Address

- Object Pointer (oop_type)
  - Object allocated in GC; May be changed during the runtime.
- Metadata Pointer (metadata_type)
  - Class and Method data including profile data, bytecodes, and constants.
  - Dynamically loaded by java.lang.ClassLoader
- Address inside the JVM Runtime (static_call, virtual_call, runtime_call, external_word)
  - Stub Routines (aka Runtime specific subroutines) e.g. arraycopy, sin(float)
  - Internal Tables
  - String Message

Object Pointers

Constants

- Most object pointers used in jitted code are constants (aka static final variable).
  - For example:
    ```
    private static final Unsafe UNSAFE = Unsafe.getUnsafe();
    ```
  - java.lang.String Constants are also static final.
  - Array Elements in a static final field are also considered static final.
  - java.lang.Class Instances are also considered static final.

Object Pointers

Trusted non-static fields

- Trusted non-static final fields could also be used as object pointers in jitted code. Currently, we just disable this feature.
    - Trusted non-static final fields are mostly variables that will be set at the start of the program (the boot layer), and remain constant during user's code.
    - Trusted non-static final fields are defined in the following function.

```
bool trust_final_non_static_fields(ciInstanceKlass* holder)
```

Object Pointers

Pre-defined Exceptions

- There are a series of exceptions used in jitted code.

```
ciInstance* NullPointerException_instance();
ciInstance* ArithmeticException_instance();

// Lazy constructors:
ciInstance* ArrayIndexOutOfBoundsException_instance();
ciInstance* ArrayStoreException_instance();
ciInstance* ClassCastException_instance();

ciInstance* the_null_string();
ciInstance* the_min_jint_string();
```

Object Pointers

Current State

- I have fully viewed related c1 compiler’s code. And I’m sure that all possible object pointers are correctly handled.
- However, reading C2 compiler’s code is not an easy task (It use DFA and code generation). We may checking all possibility of object pointers in the future.

Metadata Pointer

Metadata

- What is a Metadata? There are 5 classes that inherit from Metadata:
  - Klass: Inner representation of a java Class data. (Constants, Fields, Methods)
  - ConstantPool: Constants in a specific class defined in Bytecode.
  - Method: Inner representation of a method data. (Name & Signature, Profile information, Code entry, etc)
  - MethodCounters: invocation counter & backedge counter in a method. Mainly used in interpreter state (compile level 0) and limit profile collection state (compile level 2).
  - MethodData: All profile information in a method. Including counters, branch counters, virtual call types. Mainly used in full profile state (compile level 3).

Metadata Pointer

Metadata and Classloader

- Metadata (Klass/Method/ConstantPool) need to be load from an instance of java.lang.Classloader, which is dynamically defined during the runtime.
- Currently we just skip loading the relocated jitted code if current java.lang.Classloader can't find a class.

- Note: Class-loading during compilation is disabled by JVM, and I enable that feature. Is that a good practice?

Metadata Pointer

MethodCounters & MethodData

- MethodCounters:
    - Limited profile information will be record in MethodCounters struct. However, JVM does not provide reloc information for MethodCounters.
    - Most MethodCounters pointers are embedded as a Constant in C1 LIR (LIR_Const). Other possibilities should be checked in the future.
- MethodData:
    - Unlike MethodCounters, JVM provides reloc information for MethodData and its relocation is easy to implement.
    - Merge Operation that merge 2 existing MethodData is harder to implement, as you need to understand every profile entries in MethodData.

Address in JVM Runtime

- JIT compiler always embed a number of inner addresses into jitted code, which is a huge problem for us. These addresses include:
  - Runtime Stub: subroutine for specific Runtime (Allocation subroutine, Exception handler, etc.)
  - Inner function: call inner c++ function directly. Such as:

    ```
    void MacroAssembler::debug64(char* msg, int64_t pc, int64_t regs[]);
    jlong os::javaTimeNanos();
    jlong   ldiv(jlong y, jlong x);
    ```

  - String Messages
  - ....

Address in JVM Runtime

Current State

- Only frequently-used addresses is considered in current implementation.
- It can cover most cases in our benchmarks.