Introduction to the algorithm used by the JMTrace

In this tool, the implementation method I used is based on the bytecode level. Specifically, the ASM framework is used to instrument the bytecode. Use the java.lang.Instrument interface provided by the JVM.
Specifically, the task in this programming job is actually trying to record the field read and write of the object and the read and write of the array elements at runtime. So at the bytecode level, we are equivalent to processing the following related instructions:

1. getstatic getfield
2. putstatic putfield
3. *aload
4. *astore

With a simple understanding of the JVM memory model, our task can be expressed as follows:
When the above instructions appear, out of trust in the implementation of javac, we can actually assume that the stack has the compliance content specified in the bytecode manual at that time. Then we can use the stack data operation instructions provided by the JVM on the basis of this known information, prepare a copy of the information we need to record on the stack as a function to pass in parameters, and call a static function for information recording "consume" the extra content we write in the stack, so that we can achieve "no side effects" instrumentation.

If take getfield as an example:
The stack changes when this bytecode runs: ..., objectref → ..., value
Then the bytecode getfield implies the memery access type "W", and the pid can be obtained by calling the system API in the recording function. To identify the unique id of the read and write object, here simply use the object, the object corresponds to the class object, and the object field's name through a hash function. In objectref in the stack, the class object can be actually known to know the name, and the specific field name can also be known in getfield. Therefore, all the information we need has been obtained.
In principle, all six bytecodes are processed in the above mode. The relatively tricky thing is actually the interference of the value written in the write bytecodes: in the JVM implementation, the long type and the double type is different from the other types. It occupies twice the space in the stack, and the other is *store bytes. The information we need is two or three slots below the top of the stack, which makes stack operations relatively interesting.

*store stack operation:
General type (not long or double)
..., arrayref, index, value
DUP_X2 (process the value deep in the stack)
..., value, arrayref, index, value
POP (remove value)

..., value, arrayref, index
DUP2_X1 (copy the top of the stack and insert it under the value to restore the scene, and prepare static invoke at the same time)
..., arrayref, index, value, [arrayref, index] ([] framed is what the function consumes)
Similarly, with a slight change, it can be changed to an adaptive width value:
..., arrayref, index, valueW
DUP2_X2
..., valueW, arrayref, index, valueW
POP2
..., valueW, arrayref, index
DUP2_X2
..., arrayref, index, valueW, [arrayref, index]

In addition, I have also made certain optimizations in terms of information recording. Taking into account that after this tool is running, the user program in the JVM will call the relevant log method every time the relevant bytecode is run. It is obviously time-consuming to piece together the output string in the log method each time. So here we do not use the immediate output strategy, but the strategy of keeping the 4 key contents in the memory. On the other hand, the program I/O is not directed to the terminal or redirected to a certain file at runtime, but is written to the memory. Such tools are obviously more efficient.
The tool sets addShutdownHook, and it officially starts output when the JVM exits.
But this strategy also has a problem: it needs extra space when running, and the program does not exit, the result is still in the memory, and it is relatively easier to lose data. But based on my personal use of TamiFlex and my experience in my graduation project, I think that the improvement of runtime efficiency in dynamic analysis is relatively more important.

Reference

[1]. The Java Virtual Machine Instruction Set https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-6.html
[2]. ASM 9.2 javadoc https://asm.ow2.io/javadoc/
[3]. https://stackoverflow.com/questions/57398474/is-there-a-way-to-swap-long-or-double-and-reference-values-on-jvm-stack
[4]. TamiFlex https://github.com/secure-software-engineering/tamiflex
[5]. Graduation Project: DyDy