JVMTIPROF: An extension to the Java Virtual Machine Tool Infrastructure for building profiling agents

Denilson das Mercês Amorim

Universidade Federal da Bahia

2022

Table of Contents

Introduction

Motivation

- Profiling commonly used to identify application bottlenecks.
- Instrumentation fundamental part of profiling infrastructure.
- Tool-building systems can be used to build profilers.
- JVMTI exists in the Java ecosystem but is too bare-bones.

Contribution

- JVMTIPROF
  - An extension to JVMTI with high-level functionality.
  - Same patterns, idioms and types – reduced learning curve.
  - Agent developers can focus more on methods than infrastructure.

Contribution

- JVMTIPROF
  - An extension to JVMTI with high-level functionality.
  - Same patterns, idioms and types – reduced learning curve.
  - Agent developers can focus more on methods than infrastructure.
- How to extend JVMTI without source-level modifications.

Background

Instrumentation

- Technique to add auxiliary code to existing programs.
- Can be *static* (i.e. compile-time) or *dynamic* (i.e. runtime).
- *Function hooking* is the act of instrumenting function boundaries.
- Profiling, program analysis, code coverage, just-in-time compilation...

Profiling

- Dynamic program analysis to measure performance.
- Typically used to identify hot paths.
- Classified in *sampling profilers* and *instrumenting profilers*.
- Can feed profiled-guided optimization algorithms.

Java Virtual Machine

- Abstract Machine.
- Executes code independently of hardware and operating system.
- Oracle's HotSpot is the reference implementation.
- Garbage-collected memory management.

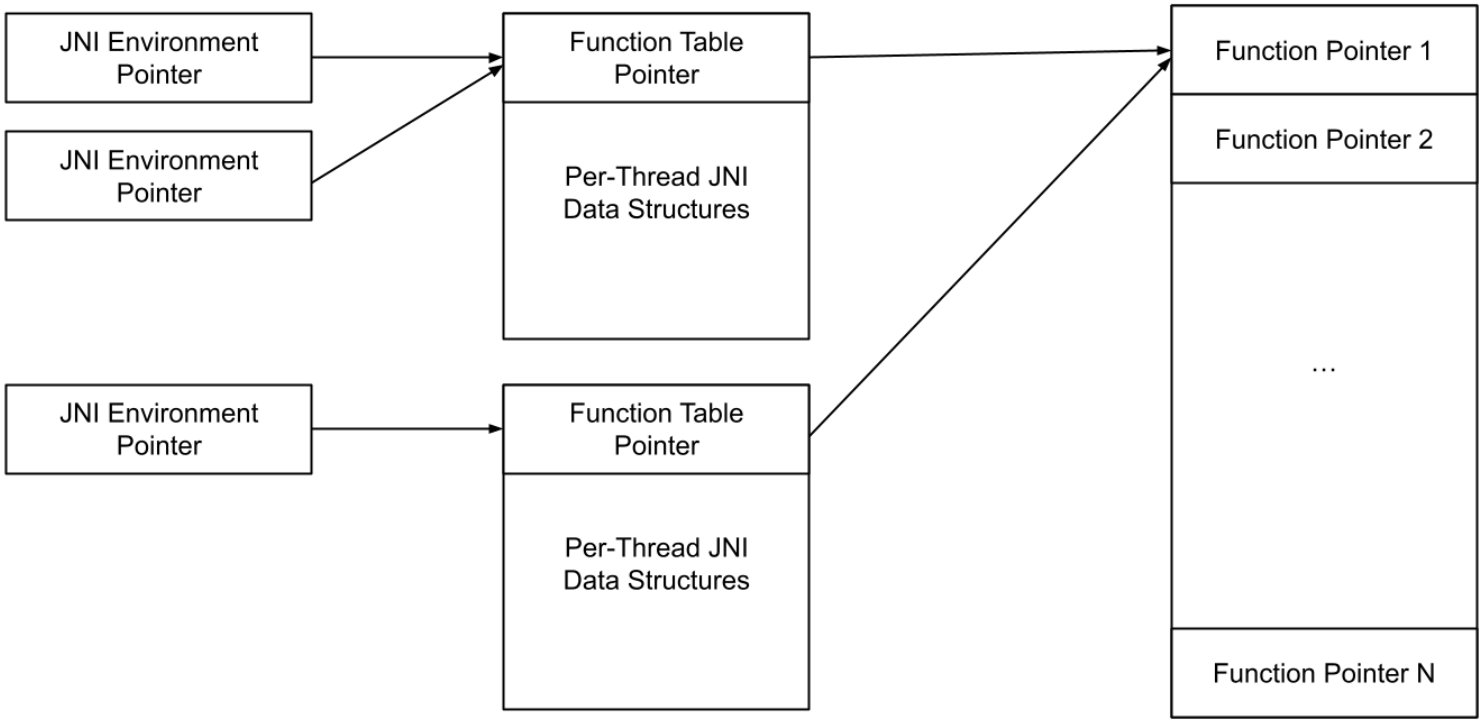
Java Virtual Machine

- Several programming languages targeting the JVM
- A program compiles into a set of *class files*.
- JVM interprets the class instructions (*bytecode*).
- Bytecode just-in-time compiled to machine code.

JVM Tool Interface (JVMTI)

- API for native agents to build development and monitoring tools.
- Native agents are written in C++.
- Can inspect VM states and react to events.
- Function table as an interface approach.
- Pay only for what you need attitude.

JVM Tool Interface (JVMTI)



JVM Tool Interface (JVMTI)

```
#include <jvmti.h>

JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm,
             char *options,
             void *reserved)
{
    jvmtiEnv *jvmti;

    vm->GetEnv((void **)&jvmti, JVMTI_VERSION);

    jvmtiCapabilities caps;
    memset(&caps, 0, sizeof(caps));
    caps.can_generate_method_entry_events = true;
    jvmti->AddCapabilities(&caps);

    jvmtiEventCallbacks callbacks;
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.MethodEntry = OnMethodEntry;

    jvmti->SetEventNotificationMode(
            JVMTI_ENABLE, JVMTI_EVENT_METHOD_ENTRY, NULL);
    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));

    return 0;
}
```

Java Virtual Machine – Safepoint Bias

- JIT-compiled code can only know root of objects through GC maps.
- Maps created only for specific instructions – the *safepoint polls*.
- JVMTI's `GetStackTrace` requires thread to be in a safepoint.
- Thus can only sample safepoint locations – *safepoint bias*.

Related Work

- Static Instrumentation
  - BCEL (DAHM, 1999)
  - ASM (BRUNETON et al., 2002)
  - Javassist (CHIBA; NISHIZAWA, 2003)
  - SOOT (VALLéE-RAI et al., 2010)
- Dynamic Instrumentation
  - FERRARI (BINDER et al., 2007)
  - DiSL (MAREK et al., 2012)
- All of them are written in and for use within Java boundaries.

Related Work

- JNIF (MASTRANGELO; HAUSWIRTH, 2014)
  - Java Native Instrumentation Framework
  - Static instrumentation in native code.
  - Can be used from JVMTI's hooks to perform dynamic instrumentation.
- Our work is the the first fully native dynamic instrumentation framework for Java (as far as we know).

Methodology

Design

- Similar to JVMTI.
- Environment created with `jvmtiProf_Create`.
- Can perform efficient *method interception*.
- Can perform asynchronous *call stack tracing*.
- Can perform *execution sampling*.
- Pay only for what you ask for ;)

Example – Sample and Print Call Stack

```
jvmtiProfEnv *jvmtiprof;

JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *options, void*)
{
    jvmtiEnv *jvmti;

    vm->GetEnv((void **)&jvmti, JVMTI_VERSION);

    jvmtiProf_Create(vm, jvmti, &jvmtiprof, JVMTIPROF_VERSION);

    jvmtiProfCapabilities caps;
    memset(&caps, 0, sizeof(caps));
    caps.can_generate_sample_execution_events = true;
    caps.can_get_stack_trace_asynchronously = true;
    jvmtiprof->AddCapabilities(&caps);

    jvmtiProfEventCallbacks callbacks;
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.SampleExecution = OnSampleExecution;
    jvmtiprof->SetEventCallbacks(&callbacks, sizeof(callbacks));

    jvmtiprof->SetEventNotificationMode(
            JVMTI_ENABLE, JVMTIPROF_EVENT_SAMPLE_EXECUTION,
            NULL);

    jvmtiprof->SetExecutionSamplingInterval(1000000000L); // 1 second

    return 0;
}
```

Example – Sample and Print Call Stack

```
void JNICALL
OnSampleExecution(jvmtiProfEnv *jvmtiprof, jvmtiEnv *jvmti, JNIEnv *jni, jthread thread)
{
    char printbuf[256];
    jvmtiProfError err;

    const jint depth = 1;
    const jint max_frame_count = 1;
    jvmtiProfFrameInfo frames[max_frame_count];
    jint frame_count;

    // fprintf isn't async-signal-safe, thus we use write.
    snprintf(printbuf, sizeof(printbuf), "sampling...\n");
    write(STDERR_FILENO, printbuf, strlen(printbuf));

    err = jvmtiprof->GetStackTraceAsync(depth, max_frame_count, frames, &frame_count);
    if(err != JVMTIPROF_ERROR_NONE)
        return;

    for(int i = 0; i < frame_count; ++i)
    {
        snprintf(printbuf, sizeof(printbuf),
                 "\tframe %d; bci offset %d; method %p\n",
                 i, frames[i].offset, (void *)frames[i].method);
        write(STDERR_FILENO, printbuf, strlen(printbuf));
    }
}
```

Implementation

- Uses some events from JVMTI to implement its functionalities.
- Collision between JVMTIPROF and JVMTI's API client desires.
- Solution: Redirect JVMTI function table to JVMTIPROF managed functions.

Method Interception

- Achieved by instrumenting class bytecode.
- JVMTI's `ClassFileLoadHook` used for intercepting class loading.
- Intercepted method invokes JVMTIPROF which invokes API client.

Method Interception

```
SetMethodEventFlag("Example", "sum", "(II)I");
```

```
public class JVMTIPROF {
      public static native void onMethodEntry(long hookPtr);
      public static native void onMethodExit(long hookPtr);
}

public class Example {

      static final long sumHookPtr = /* determined at runtime */;

      public int sum(int a, int b) {
            JVMTIPROF.onMethodEntry(sumHookPtr);
            try {
                  return a + b;
            } finally {
                  JVMTIPROF.onMethodExit(sumHookPtr);
            }
      }
}
```

Execution Sampling

- `CLOCK_PROCESS_CPUTIME_ID` timer.
- `SIGPROF` signal.
- Interval set by `SetExecutionSamplingInterval`.
- Implementation and callback must be *async-signal-safe*.

Call Stack Trace

- `GetStackTraceAsync` – interface similar to JVMTI.
- Implemented by means of HotSpot's `AsyncGetCallTrace`.
- Async-signal-safe.
- Does not require a safepoint – bias free.
- We force the VM to generate GC maps at non-safepoints.

Evaluation

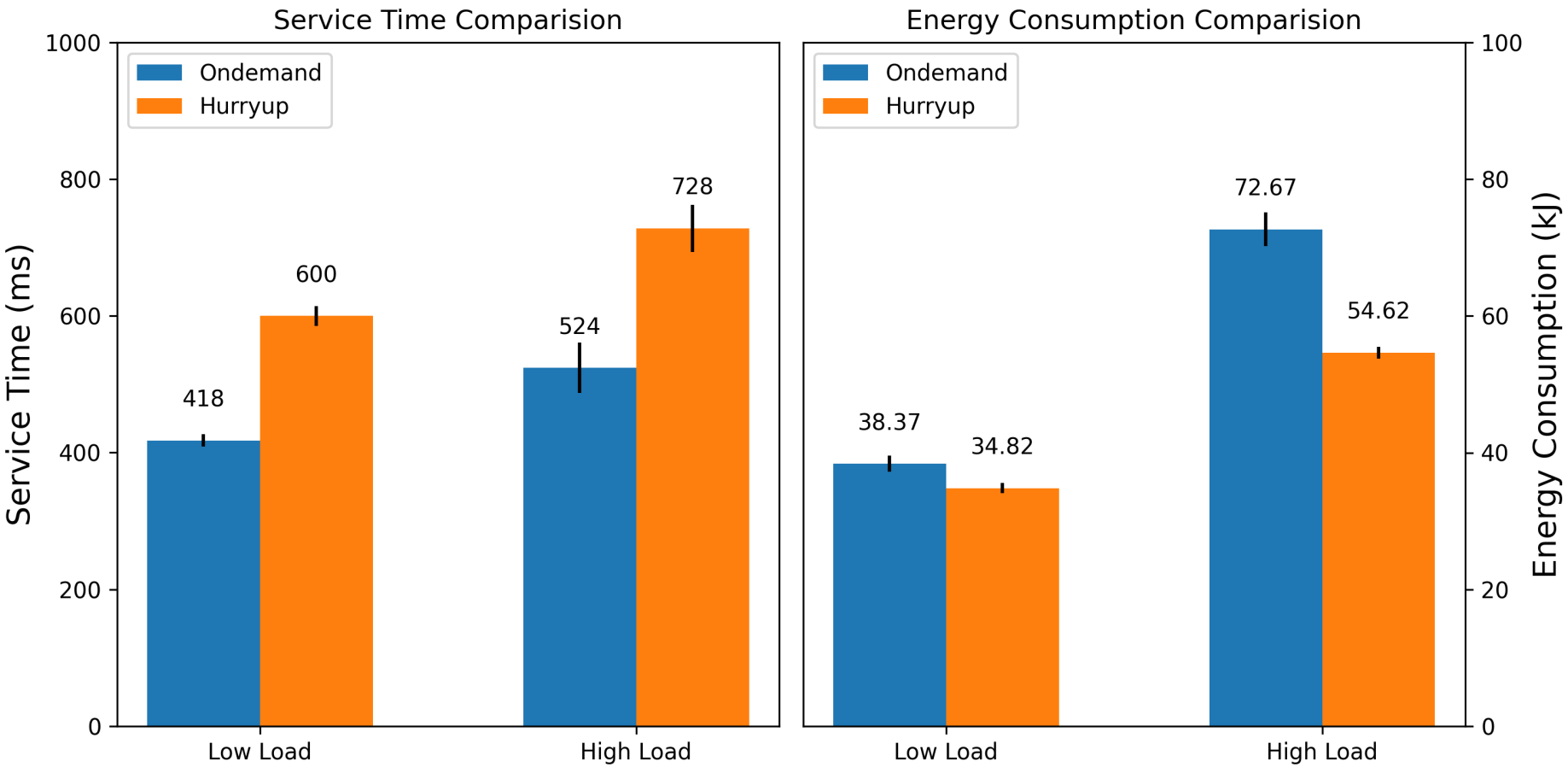
Evaluation

- Profile-guided Frequency Scaling for Latency-Critical Search Workloads (MEDEIROS et al., 2021)
  - Instruments Elasticsearch's hot functions with enter/exit events.
  - Uses these events to adapt processor core frequencies.
  - Saves energy while meeting response deadline.
  - Uses JVMTI for the job!
- What if we replace JVMTI with JVMTIPROF?

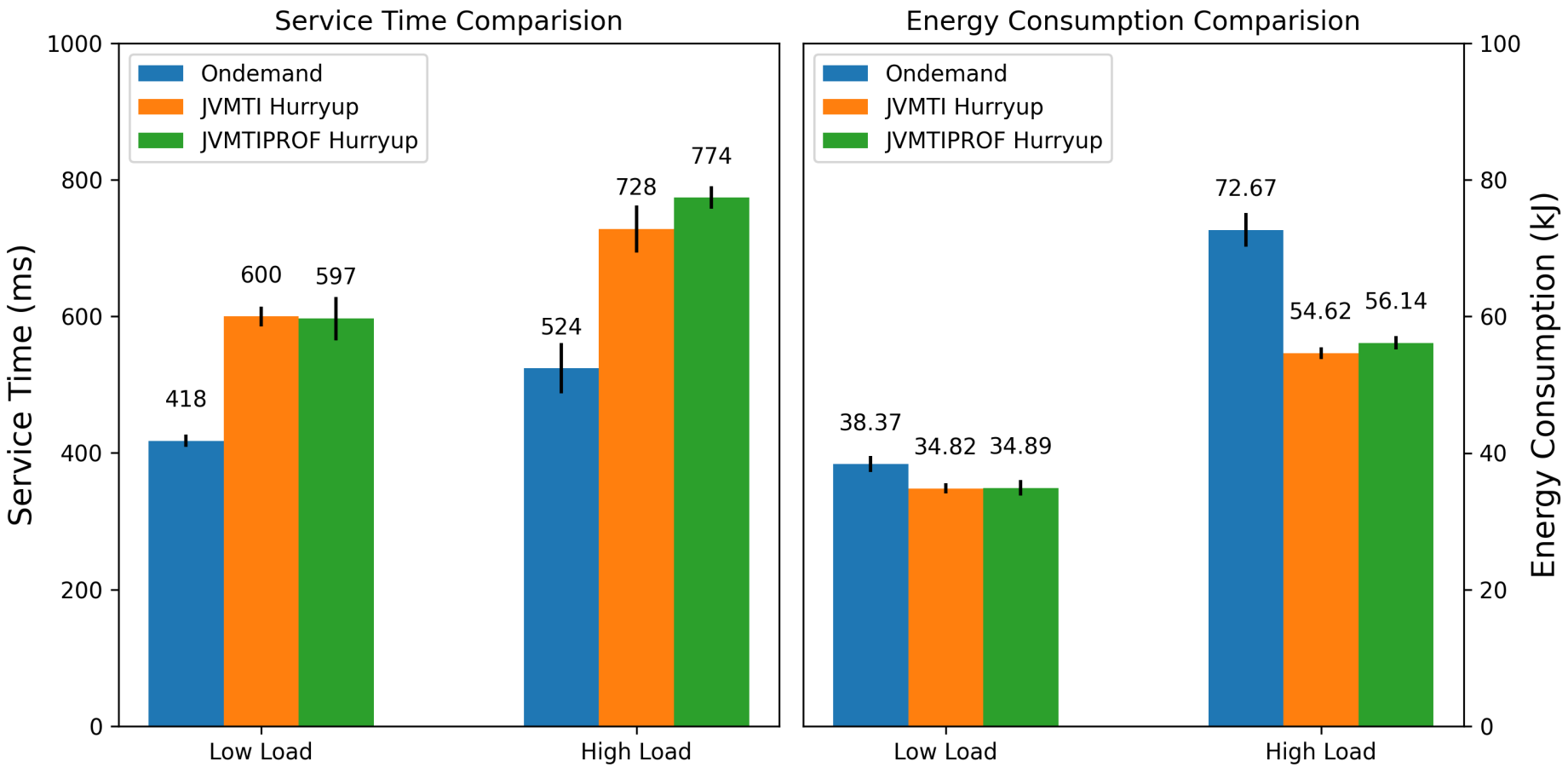
Experimental Setup

- Wikipedia indexed in Elasticsearch.
- Load-generation runs for 20 minutes, six times.
  - Low Load: 4 clients.
  - High Load: 12 clients.
  - Each client issues one request per second.

Baseline


Service Time Comparision
Service Time (ms)
1000
800
600
400
200
0
Ondemand
Hurryup
418
600
524
728
Low Load
High Load
Energy Consumption Comparision
Energy Consumption (kJ)
100
80
60
40
20
0
Ondemand
Hurryup
38.37
34.82
72.67
54.62
Low Load
High Load

Results


Service Time Comparision
Ondemand
JVMTI Hurryup
JVMTIPROF Hurryup
Service Time (ms)
1000
800
600
400
200
0
418
600
597
524
728
774
Low Load
High Load
Energy Consumption Comparision
Ondemand
JVMTI Hurryup
JVMTIPROF Hurryup
Energy Consumption (kJ)
100
80
60
40
20
0
38.37
34.82
34.89
72.67
54.62
56.14
Low Load
High Load

Concluding Remarks

Concluding Remarks

- Minimal performance penalties compared to JVMTI.

Concluding Remarks

- Minimal performance penalties compared to JVMTI.
- Extending JVMTI
  - ...by manipulating its function table is hard to maintain.
  - Alternative: JVMTI environment private to JVMTIPROF.

Concluding Remarks

- Minimal performance penalties compared to JVMTI.
- Extending JVMTI
  - ...by manipulating its function table is hard to maintain.
  - Alternative: JVMTI environment private to JVMTIPROF.
- Future Work
  - Cross-platform support.

Concluding Remarks

- Minimal performance penalties compared to JVMTI.
- Extending JVMTI
  - ...by manipulating its function table is hard to maintain.
  - Alternative: JVMTI environment private to JVMTIPROF.
- Future Work
  - Cross-platform support.
  - Additional functionalities (e.g. capture arguments).

Concluding Remarks

- Minimal performance penalties compared to JVMTI.
- Extending JVMTI
  - ...by manipulating its function table is hard to maintain.
  - Alternative: JVMTI environment private to JVMTIPROF.
- Future Work
  - Cross-platform support.
  - Additional functionalities (e.g. capture arguments).
  - Hybrid call stack tracing (PANGIN, 2016).

Concluding Remarks

- Minimal performance penalties compared to JVMTI.
- Extending JVMTI
  - ...by manipulating its function table is hard to maintain.
  - Alternative: JVMTI environment private to JVMTIPROF.
- Future Work
  - Cross-platform support.
  - Additional functionalities (e.g. capture arguments).
  - Hybrid call stack tracing (PANGIN, 2016).
  - Method interception by generating specialized code.

Concluding Remarks

- Minimal performance penalties compared to JVMTI.
- Extending JVMTI
  - ...by manipulating its function table is hard to maintain.
  - Alternative: JVMTI environment private to JVMTIPROF.
- Future Work
  - Cross-platform support.
  - Additional functionalities (e.g. capture arguments).
  - Hybrid call stack tracing (PANGIN, 2016).
  - Method interception by generating specialized code.
  - Execution sampling with a per-thread timer.

Thank You!