The Two Cultures of Programming Language Implementation

Stephen Kell
King’s College London
srkell@acm.org

July 2021
Figure 1. Internal observation by instrumentation (top) versus JPDA-based external observation (bottom).

We note that often, as with DiSL, the instrumentation itself is done in a separate process spawned by the agent.

Like physical systems, software systems exhibit a tension between observability and performance. We first discuss various practicalities of bytecode instrumentation, using JVMTI (or, rarely, performing the instrumentation offline). We call this internal observation instrumentation: its documentation notes that “goes to great pains to avoid the execution of any code in the debuggee virtual machine” because deoptimisation “causes the tool author’s problem. . .”

In-process analysis “interferes with the behavior being analysed. . . for example: . . . competition for resources becomes the tool author’s problem. . .” Avoiding perturbation therefore “controlled by a separate process which implements the bulk of the analysis’s normal execution”. Avoiding perturbation therefore approaches perturbation differently.

JVMTI allows construction of tools by linking a native library, called an “agent”, into the VM. This library interposes mechanisms offered by the JVM platform—JVMTI and JPDA—sign of VM-level observation mechanisms. The two mechanisms differ in design: remote processes require marshalling and copying code, with its associated development and runtime overheads. In contrast, processing within local instrumentation does not incur these overheads, and benefits from JIT optimisations. In this section we review a series of problems encountered while building and using DiSL, which we believe are inherent to internal observation on today’s JVM.

Most dynamic analysis tools for the JVM work by bytecode instrumentation: its documentation states that “this interface does not include some events that one might expect. . .” instead provides support for bytecode instrumentation: its documentation notes that “JVMTI’s design deliberately emphasises tool construction. . .”

Unfortunately, even simple instrumentations exhibit subtle complications; certainly, these motivated DiSL’s initial single-process design. Simplicity and performance are likely the driving forces of any tool and are absolutely central to DiSL. However, the single-process design had a disadvantage, namely, that the analysis and its instrumentation had to be coupled. We plan explanations; certainly, these motivated DiSL’s initial single-process design. Simplicity and performance are likely the driving forces of any tool and are absolutely central to DiSL. However, the single-process design had a disadvantage, namely, that the analysis and its instrumentation had to be coupled. We plan to explain this coupling, where replacement code may be supplied by the agent.

...
Problems:

- fixed set of debugging features/commands
- gaining source-level view means either
  - disable optimisation! (slow)
  - *deoptimise* on demand (hard; still somewhat slow)

Benefits:

- non-debug runs can go fast!
- VM implementer has control of complexity level
Generalise the ‘server’ to an extreme
The JVM tool interface (JVMTI) is a native programming interface for use by tools... [supporting] the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools. ... This interface does not include some events that one might expect... [but] instead provides support for bytecode instrumentation

https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html
Problems:

- JVMTI is a lot of work to use well
- its view is inherently incomplete
- it is ‘impossible’ to use reliably + completely
The JVM is Not Observable Enough (and What To Do About It)

Stephen Kell        Danilo Ansaloni        Walter Binder
University of Lugano
firstname.lastname@usi.ch

Lukáš Marek
Charles University
lukas.marek@d3s.mff.cuni.cz

The Java Virtual Machine (JVM) is the target of many
developers working with a virtual machine (VM) depend

1. Introduction

Bytecode instrumentation is a preferred technique for build-

Keywords
Measurement, Reliability, Performance

General Terms
Languages

Categories and Subject Descriptors
D.3.4 [Categories and Subject Descriptors]: Processors—run-time environments

Abstract

We argue that no Java platform mechanism provides simul-

tered while building the DiSL instrumentation framework.

We illustrate its dangers with several examples gath-

The JVM is Not Observable Enough (and What To Do About It)
Stephen Kell Danilo Ansaloni Walter Binder
University of Lugano
firstname.lastname@usi.ch

Lukáš Marek
Charles University
lukas.marek@d3s.mff.cuni.cz

The JVM is Not Observable Enough (and What To Do About It)
Stephen Kell Danilo Ansaloni Walter Binder
University of Lugano
firstname.lastname@usi.ch

Lukáš Marek
Charles University
lukas.marek@d3s.mff.cuni.cz
Problems:
- JVMTI is a lot of work to use well
- its view is inherently incomplete
- it is ‘impossible’ to use reliably + completely

Root problem:
- observation is ‘internal’, *within* the VM’s guest program
- . . . sharing VM state – no isolation

Root benefit:
- can go fast! bytecode optimised as a unit
- VM implementer has little to do
‘Keep it in a language virtual machine’ goes way back. . .

“[c. 1959] S.R. Russell noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter.”

John McCarthy
History of Lisp
HOPL, 1978
A different approach to observation. . .

(. . . taken by almost no JVMs, but. . . )
$ cc -g -o hello hello.c && readelf -wi hello | column

<b>:TAG_compile_unit

AT_language : 1 (ANSI C)
AT_name : hello.c
AT_low_pc : 0x4004f4
AT_high_pc : 0x400514

<7ae>:TAG_pointer_type

AT_byte_size: 8
AT_type : <0x2af>

<76c>:TAG_subprogram

AT_name : main

<c5>: TAG_base_type

AT_byte_size : 4
AT_encoding : 5 (signed)
AT_name : int

AT_type : <0xc5>
AT_low_pc : 0x4004f4
AT_high_pc : 0x400514

<2af>:TAG_pointer_type

AT_name : argc
AT_type : <0xc5>
AT_location : fbreg - 20

<2b5>:TAG_base_type

AT_byte_size: 1
AT_encoding : 6 (char)
AT_name : char

AT_type : <0x7ae>
AT_location : fbreg - 32
In most ahead-of-time toolchains, observation is external
- isolated – no shared state
- . . . or knowledge! VM describes its own implementation
- . . . via extensible metadata (+ OS APIs only)

Trade-offs:
- view is inherently limited by metadata
- generated imperfectly by compilers
- fine for exposing state; run-time events trickier

How it pans out:
- forces choice between speed and debug fidelity
- debugged program is always ‘full-speed’!
- compiler author can ‘tailor’ their effort/reward
The toolchain approach also goes way back.
“Neither culture knows the virtues of the other.”

I sometimes feel this way about virtual machines versus toolchains.

Most engineers/researchers become expert in only one.

That is a generalisation. . . we are not that bad. (And Snow’s piece is not that good. . . )

Still, we could do better. Better at what? Well. . .
VMs tend to prioritise the humane, outwardly. Internally, modern production VMs are some of the most complex and subtle codebases around.

Toolchains present as workmanlike, but tend to be ‘tinkerable’ at the edges—they have more edges

It’s not ‘VMs human-first; toolchains technology-first’.

It’s ‘both started humans-first, but are now technology-first.’
In Smalltalk’s “integrated environment”... there is little distinction between the compiler, interpreter, browser and debugger, [all of which] cooperate through shared data structures. . . . Pi is an isolated tool in a [Unix] “toolkit environment” [and] interacts with graphics, external data and other processes through *explicit interfaces*.

T.A. Cargill

*Pi: a case study in object-oriented programming*

OOPSLA ’86
The ‘best of both’ story

For ‘computers that work for humans’, maybe we need ‘best of both’?

Certainly there is room to

- bring VM-like techniques to native toolchains
- and vice-versa

(... insert adverts for some past work by me and others)

But something is wrong with this picture. Mostly, we *don’t*!
has been the widespread realization that the usefulness of such a system is critically dependent on the quality of the software provided to facilitate the interaction between user and machine. In particular, one area of critical importance for effective utilization of such a system is that of facilities for program debugging. In view of the important role they play, surprisingly little attention has been paid to the development of facilities to aid in the process of on-line program debugging. Furthermore, much of the work in this field has been described only in unpublished reports or passed on through the oral tradition, rather than in the published literature. The purpose of this paper is to summarize the existing work in this area and
Your Debugger Sucks

TL;DR Debuggers suck, not using a debugger sucks, and you suck.

If you don't use an interactive debugger then you probably debug by adding logging code and rebuilding/rerunning the program. That gives you a view of what happens over time, but it's slow, can take many iterations, and you're limited to dumping some easily accessible state at certain program points. That sucks.

If you use a traditional interactive debugger, it sucks in different ways. You spend a lot of time trying to reproduce bugs locally so you can attach your debugger, even though in many cases those bugs have already been reproduced by other people or in CI test suites. You have to reproduce the problem many times as you iteratively narrow down the cause. Often the debugger interferes with the code under test so the problem doesn't show up, or not the way you expect. The debugger lets you inspect the current state of the program and stop at selected program points, but doesn't track data or control flow or remember much about what happened in the past. You're pretty much stuck debugging on your own; there's no real support for collaboration or recording what you've discovered.

Software developers and companies everywhere should be very sad about all this. If there's a better way to debug, then we're leaving lots of productivity — therefore money — on the table, not to mention making developers miserable, because (as I mentioned) debugging sucks.

If debugging is so important, why haven't people built better tools? I have a few theories, but I think the biggest reason is that developers suck. In particular, developer culture is that developers don't pay for tools, especially not debuggers. They have
We have two cultures, and they both stink!

Culture is, roughly, the set-difference between what we could do, and what we actually do.

(Existence of a difference is not a negative; it’s necessary.)

What we do is worry about small-% performance deltas.
  - both our approaches to observation were boxed in by performance

What we don’t do, or not to the extent our forebears did, is cling to lofty ideals about serving humanity.
  - somehow we are kept busy even in absence of that
Some established beliefs in computer science

My take: all the following are false! Many (not all) were initially true.

- software performance is an infrastructure problem
- progress comes mostly from designing new things
- interfaces are the path to modularity
- productivity is about writing more code
- reliability can largely be addressed by mathematical proof
“Work expands so as to fill the time available for its completion.”

C. Northcote Parkinson

*Parkinson’s Law*

John Murray, 1958

Software performance is now a social and cultural problem, not an infrastructure problem (in >99% of cases).

(Infrastructure performance improvement is mostly good for helping giant companies save money on their electricity bills.)
Induced demand
I’m not sure our culture has caught on

In research:

VMIL ’17: panel on ‘the future of language runtimes’ . . . focused 100% on performance

ICOOOLPS ’19 interactive session: ‘hardware is slow’

ICOOOLPS ’19 Project Snowflake talk: case study in the ‘extreme economics’ of huge companies

‘Extreme economics’ are talking more loudly each year (ask me)
What happened to. . .

Increasing the individual’s unit capability?

Understanding what those workloads are *really* doing?

(Observation tools are more important than ever!)

What if. . . ?

- record/replay were standard (in the OS)?
- minimalism were fêted like *scale* is?
- integration received as much attention as performance?
Picture credits

John McCarthy – unknown; likely Stanford University

Snow first edition – Oxford University Press (fair use)

Children with Alto – PARC (fair use)

Ken and Den at the PDP-11 – Peter Hamer (CC-BY)

Parkinson first edition – John Murray publishers (fair use)

traffic – public domain (US EPA, May 1972)

Ken and Den (Jargon File): public domain
Basic steps—thoughts in progress

Form community
- “academically monetise”
- where relevant, be up-front about incentives, power, . . .

Hacks
- target domains that defuse typical performance expectations
- “measure other stuff” and do a good job (e.g. code complexity?)
- collaborate outside core CS, on ‘getting stuff done’ e.g. sci. comp.

Needs thought
- tech transfer models
- licensing & business models for software
- (basic research on integration, deployment, . . . can feed in here)
Technical openings

Even cheap devices are insanely powerful
- use the hardware bounty *against* the status quo
- touchstone: RPi ‘data centres’; AWB’s ‘personal habitats’…
- SSI distributed OS between your smartphone & router?

Commodity software wants to collapse under its own weight.
- i.e. increasingly unworkable-on, unworkable-with
- touchstone: Viewpoints’s STEPS project (wanted: less clean-slate)
- economic incentives may be there, just seeking an alternative

Create systems (involving languages), not ‘just’ languages/impls
- systems have applications. . .
- . . . and can be evaluated w.r.t. those
- ‘performance’, like ‘language’, often abstracts away the application

Can’t recreate world in our image, but can ‘swim towards’ our vision
Most people in this room have some agency

- what do I research? what do we fund?
- what expertise do we hire? who do I work for?
- what business models do we pursue?
- how do we conceive of ourselves? what’s our *culture*?
Parkinson’s Law in detail

(1) “An official wants to multiply subordinates, not rivals.”
- i.e. local incentives to ‘grow the stack taller/deeper’
- but abstraction isn’t globally free, w.r.t. system complexity

(2) “Officials make work for each other.”
- i.e. integrating $n$ components has superlinear cost

Avenue 1: radically reduce integration-related overheads
- not studied to much depth, in CS history today

Avenue 2: avoid tendencies to taller towers of abstraction...
- suspect integration is key here too
- Unix filesystem example
ENERGY FORECAST

Widely cited forecasts suggest that the total electricity demand of information and communications technology (ICT) will accelerate in the 2020s, and that data centres will take a larger slice.

- Networks (wireless and wired)
- Production of ICT
- Consumer devices (televisions, computers, mobile phones)
- Data centres

The chart above is an ‘expected case’ projection from Anders Andrae, a specialist in sustainable ICT. In his ‘best case’ scenario, ICT grows to only 2.5% of total electricity use by 2030, not the anticipated 20.9%.
Proebsting’s law (http://proebsting.cs.arizona.edu/law.html)

“compiler optimization advances double computing power every 18 years”

... 

“Perhaps this means Programming Language Research should be concentrating on something other than optimizations.”

(+ remember software still gets slower! i.e. this ‘doubling’, like Moore’s, is ignoring the effects of Parkinsonian induced demand.)
Consider engineers attempting to diagnose a serious outage under time pressure—every second spent trying to understand the language is one not spent understanding the problem.

The rhetoric is crafty, but the proposals benefit parties for whom

- stakes are big \textit{per second}
- a rapidly-changing language \textit{helps} understanding (!)
- engineering resource is no big deal
  - forward-porting sources is trivial
  - no binary compatibility, no problem

i.e. large teams, mighty infrastructure, . . .
Humans named Ken and Den

“Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements... we considered this increase in size quite acceptable.”

Dennis M. Ritchie and Ken Thompson
The Unix Time-Sharing System
CACM 17(7), July 1974