Comparing execution traces of programs written in different programming languages

Jaka Bac, Boštjan Slivnik[1]

[1]*Faculty of Computer and Information Science, University of Ljubljana, Slovenia*
Email: jaka.bac@gmail.com

Abstract

This is a preliminary study about how, when executed, programs that are performing the same computation but are written in different programming languages produce different execution traces, i.e., streams of executed instructions. Until proven otherwise, any such study must be considered both CPU and compiler/interpreter dependent. The study is so far limited to the x86-64 architecture and to a small set of programming languages, i.e., C, Java and Python, that differ significantly in terms of an execution at the binary level. The comparison was carried out by analysing the execution traces obtained by the dynamic binary instrumentation of programs: when instructions found in execution traces are categorised, a difference between those three programming languages are visible from yet another point of view.

1 Introduction

Algorithms were started to be analysed soon after they have been first conceived and programs once they were written. Different approaches exist, from source code symbolic analysis to measuring CPU or wall clock time, each of them telling one part of the story.

Knuth realised that the language significantly influences the way programs are written *while machine-oriented language ... is much closer to reality* and therefore uses the assembly language to explain and analyse algorithms [1]. To bring the investigation on how programs written in different high-level programming languages are executed *closer to reality*, the execution traces, i.e., the sequence of instructions executed by the CPU once the program is run, must be analysed.

The idea of producing the execution trace of a running program is not new. But in 1990, for instance, it was established that the *additional code required to record events greatly slows a program's execution* and that *the resulting trace files can grow unmanageable large* [2]. Even in 2005, the execution traces had to be compressed on-line, an approach than severely complicated any post-mortem analysis [3]. Nowadays, with significantly increased computer resources, at least some analyses are becoming feasible and are already being performed by commercial software. Security researchers use dynamic instrumentation as an aid to discover flaws in programs. Reverse-engineers use it to discover data and control flow of an application. Profiling tools use it to detect memory access errors and/or measure performance. Furthermore, some virus scanners perform a "deep screen" of an unknown application by running it under its own dynamic instrumentation engine to detect suspicious behaviour [4].

This paper is organised as follows. Sections 2 and 3 contain a very short introduction to code instrumentation and recording of execution traces in Intel PIN. In Section 4 a simple analysis of instruction categorisation is introduced while Section 5 presents the report on performing the categorisation on a few simple programs.

2 Code instrumentation using PIN

To produce an execution trace an application must be orchestrated with small fragments of trace generating code. Instrumentations can be done at various stages, each of them having different strength and limitations.

Static instrumentation can be performed in compile time or later, but before the program is executed. If performed at compile time, it can use all information about the program available to the compiler and tends to have slightly lower overhead than run time instrumentation, but is usually unable to handle self modifying or runtime-generated code [5].

Dynamic instrumentation is done in run-time. It tends to be far more flexible in what can be instrumented. For instance, machine code generated in run-time by just-in-time compilation can only be instrumented using dynamic instrumentation. However, it may have higher performance overhead [5]. Furthermore, the implementation of dynamic instrumentation is significantly more demanding that implementation of static one.

Fortunately, various free or open-source frameworks for the dynamic instrumentation are available. Frida is an interactive executable exploration tool allowing the user to instrument the executable by injecting javascript code. Valgrind first transforms the executable into its own IR (called VEX) upon which transformations and analyses are performed. It is therefore unsuitable in our case. DynInst was not explored since DynamoRIO and PIN seem to be far more widely used. Finally, Intel PIN has been chosen as it has the most versatile API [6, 7].

At its core Intel PIN is a virtual machine with a JIT compiler which transforms existing machine code into equivalent machine code which allows PIN to retain con-

trol while the application is being executed. During the execution, PIN splits application's code into traces which are straight lines of application code that either unconditionally terminate, have a pre-defined number of conditional exit points or contain a pre-defined number of instructions. PIN rewires the exit points back into its VM to regain control and to find the next trace of application code [6].

To facilitate code instrumentation PIN provides a way to write plugins (called Pintools) which register interest for certain events and then receive callbacks from the Pin VM. There are two callbacks meant for implementing custom instrumentation. One occurs each time PIN visits a new machine instruction of the target application or when it discovers a new trace of application code. The trace is conveniently split into basic blocks for the Pintool which then uses the PIN API to either just inspect the application code and/or rewrite the application code by inserting new instructions and/or deleting existing ones.

The Pintools are written in C/C++ and must respect PIN's rules in order not to crash the instrumented application or cause a deadlock.

3 Recording execution traces using PIN

Our trace generating code is a Pintool which processes each basic block discovered by PIN VM. It disassembles its instructions and assigns them a sequential id. Optionally, the disassembly may contain the address of each instruction. Each basic block is also instrumented with a call to a tracing function which accepts the id of the basic block as a parameter in order to retrieve its disassembly and writes it to a file or a pipe. It is also possible to write out the instruction in their binary form in order to save space. The resulting trace can be processed further for more in-depth analyses. These can be written in any high level language and are not restricted by the rules for Pintools.

Once the interesting behaviour is isolated a specialised Pintool can be written in order to minimise the instrumentation overhead. A significant amount of overhead represents writing the trace to a file. If this can be reduced by performing more analysis during the program execution under PIN, a significant speedup can be obtained.

4 Code orchestration for instruction categorisation

The first analysis one can imagine when performing online analysis of an executed trace is to categorise the instructions and see whether there is any significant difference when a program is written in one programming language or another.

To speed up the instrumentation the trace-producing Pintool was extended with an option to count and categorise the instructions during the execution and to compute how much space would a disassembly trace (with addresses) take. It uses the same categorisation of instructions as Intel XED [8], a disassembler built into Pin (but also available as a separate project).

In order to isolate the part of the program which performs the actual computation from the startup and I/O we also developed a DLL exporting two functions which serve as markers marking the beginning and the ending of the interesting region, respectively. When the program is instrumented by our Pintool these functions are detected (by using export information) and replaced by routines in the Pintool itself which toggle a flag signalling the tracing routine that the interesting region of code is executing.

Furthermore, system libraries and C/C++ runtime code are also filtered out. For C/C++ only the program executable is traced. Only python.exe and python27.dll were traced for Python. For Java, java.exe, jvm.dll, java.dll and verify.dll were traced.

Isolation is performed by providing a list of modules to be filtered to our Pintool. All code belonging to the address space of the module is executed in the original form, but unfortunately it can not be executed at native speed since all code must pass through Pin's VM so it can maintain control of the program.

The Pintool can also differentiate between code generated at runtime (for example by JVM JIT) and code loaded from the executable images. All code which does belong to a section of an executable (or dynamic library) image is considered as dynamically generated.

At the end of execution the Pintool produces a table specifying the instruction count by category, marked/unmarked and if it is dynamically generated. The size of the disassembly trace is computed simply by summing the lengths of disassembled basic block instructions that would be written into a file as a trace.

Finally, the entire code is available at *https://github.com/JakaBac/TraceTool* and *https://github.com/JakaBac/marker/tree/master*.

5 Experimental results

To test the described orchestration four tests were devised, each based on one the following four well-known algorithms:

- the standard matrix multiplication (ijk- and ikj-multiplications),
- Strassen matrix multiplication (running without escaping to standard matrix multiplication when the size of the matrix becomes small),
- heapsort (iterative implementation) and
- quicksort (recursive implementation).

These test programs were written in C++, Java, and Python. In all three cases, the most standard language tool available on MS Windows was used for each language: VisualC++, Oracle's Java SDK and CPtyhon, respectively.

For every test and for every language, i.e., C++, Java, and Python, the ratio between the wall-clock time of the plain executable and the PIN-orchestrated executable was measured. The results are presented in Table 2. The size of execution traces are presented in Table 3. In all matrix multiplication tests matrices of 100×100 were used. In all sorting tests tables of 100000 elements were used.

TESTCASE	1×C++	10×C++	1×Java	10×Java	1×Python	10×Python
MtxMul ijk	1.00	10.00	34.11	61.89	128.80	1288.15
MtxMul ikj	1.00	2.47	101.85	128.66	238.32	2383.47
Strassen	1.00	10.00	199.50	1784.97	128.25	1282.82
Heapsort	1.00	10.09	4.12	21.79	145.90	1459.09
Quicksort	1.00	10.07	15.34	27.82	90.81	915.84

Table 1: The number of instructions executed by each algorithm if run 1 or 10 times in a row relative to the number of instructions executed by a C++ algorithm.

TEST CASE	no PIN	with PIN	ratio
$\text{MtxMux}_{\text{ikj}}$ C++	0.052	1.962	37.731
Java	0.192	31.961	166,464
Python	0.258	27.613	107.027
Strassen C++	0.061	2.530	41.475
Java	0.674	79.833	118.447
Python	0.394	46.131	117.084
Heapsort C++	0.069	1.776	25.739
Java	0.918	80.823	88,042
Python	0.901	97.440	108.147
Quicksort C++	0.057	1.703	29.877
Java	1.073	82.904	77.264
Python	0.357	33.513	93.874

Table 2: The wall-clock time of test programs: every program written in C++, Java and Python is measured when running with and without PIN.

TEST CASE	trace size
$\text{MtxMux}_{\text{ikj}}$ C++	446.981 Mb
Java	27024.72 Mb
Python	49886.81 Mb
Strassen C++	1036.11 Mb
Java	148495.66 Mb
Python	100014.22 Mb
Heapsort C++	1808.58 Mb
Java	91581.89 Mb
Python	235519.38 Mb
QuicksortC++	911.54 Mb
Java	90099.81 Mb
Python	74920.26 Mb

Table 3: The size of execution traces produced by the test programs written in C++, Java and Python.

It follows from Tables 2 and 3 that producing and analysing the execution traces of certain algorithms is manageable even if the execution traces must be stored on a disk. If the analysis can be performed on-line, one can disregard the trace size and take the ratio between running times of unorchestrated and orchestrated executables as an indication that in most cases analyses can indeed be carried out.

When comparing the instruction count of the same program between different languages the patterns illustrated by the data in Table 1 can be observed. When the problem is increased by a factor 10 then the Python instruction counts also increase by approximately the same factor. C/C++ generally shows same behaviour, but in some cases the factor is quite different. For example in the MtxMult program (using IKJ indexing) the compiler is able to vectorise the inner loop (compiler was allowed to utilise AVX2 instruction set). By doing so, instruction count is only 2.479 times bigger. It is also interesting that the compiler chose to vectorise the loop only when 10 iterations was performed. With a single iteration no vectorisation was carried out. When IJK indexing is used the compiler cannot perform vectorisation in any case. It can also be observed that the JVM is performing profiling and various optimisations during the JIT compilation. In this case the number of instructions does not follow the increase of iterations. JVM did not perform vectorisation in the IKJ MtxMult case.

Once the execution traces are obtained, it is possible to categorise the instructions. Assuming the Intel's own categorisation [8], the compact representation of the categorisation of all instructions found in traces of all tests over all three programming languages is shown in Figures 1 and 2. Especially in Figure 2 it can be observed a distinctive pattern of Python programs. C++ programs have the least uniform categorisation. We attribute this to the compile-time optimisation while the effect of just-in-time compilation and optimisation performed by Java is visible (but due to the lack of space not fully exposed).

6 Conclusion

In any case, it must be emphasised that the results shown in Figures 1 and 2 cannot be understood as definite yet. More testing must be done, with more different test and different programming languages, before any definite conclusions can be made.

By looking at the differences in Figures 1 and 2 we can understand why Knuth wanted to analyse programs at machine code level and why he even invented his own assembly language for this purpose. Even with seemingly small changes to a program's source code the same compiler may make different assumptions and produce different machine code which may run on the target architecture with different level of efficiency, which then distorts high level performance measurements.

Finally, there are at least three important details that were left out of this study: the cache misses, out-of-order instruction execution, and threads. So far the information about cache misses has not been gathered while recording execution traces and it is yet to be seen if and how this could be incorporated into the investigation. Out-of-order execution is a matter of the CPU organisation and nowadays it is often not fully disclosed by the CPU manufacturer. Runtime environments of many programming

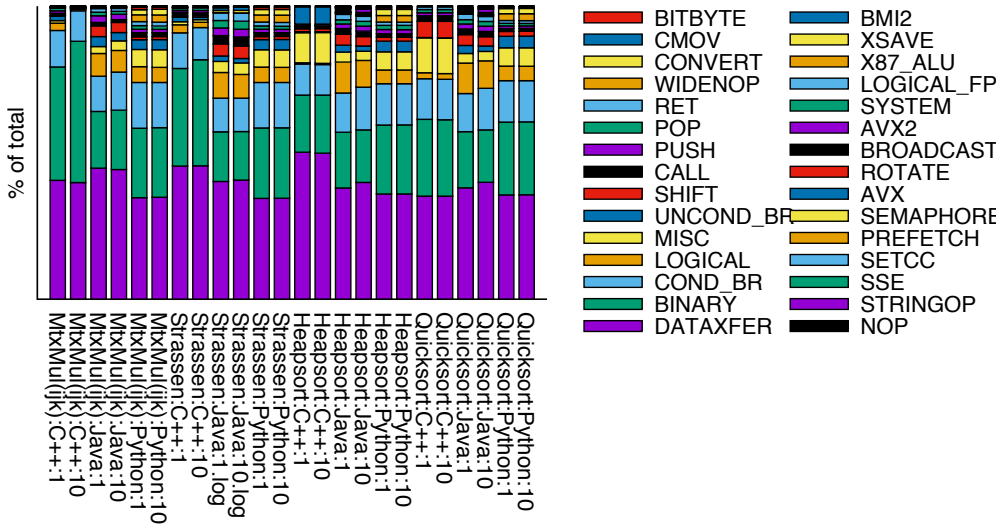

Figure 1: Instruction categorisation for various tests. Labels below the bars consists the test name, the programming language and the number of repetitions of an algorithm within the test. In all cases, most instructions executed are either data transfers (DATAXFER) or binary operations (BINARY).

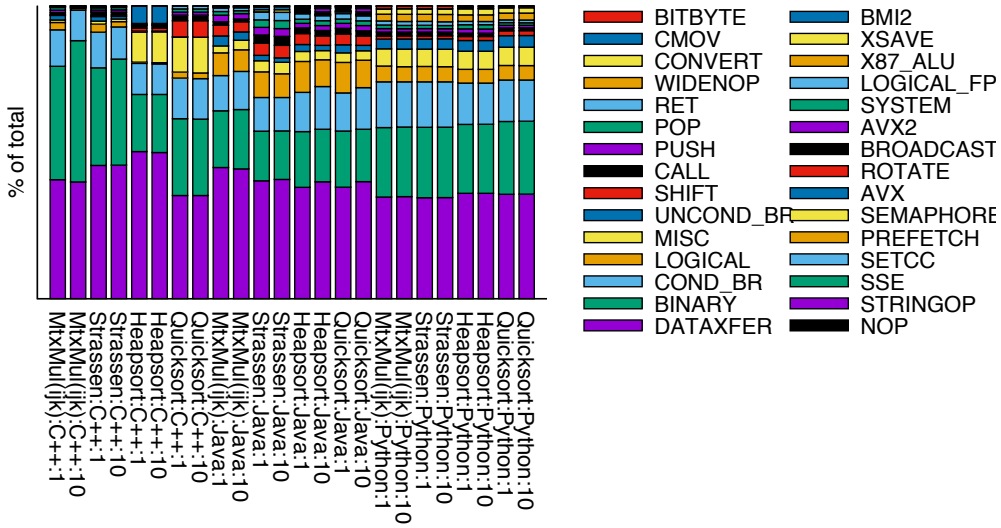

Figure 2: Instruction categorisation for various programming languages. Labels follow the same convention as in Figure 2.

languages are inherently multi-threaded. Decoupling of individual thread's traces has not been performed so far but it is planned for the future.

References

[1] D. E. Knuth, *The Art of Computer Programming*, Vol. 1 (Fundamental Algorithms), Addison-Wesley, Reading, MA, USA, 1997.

[2] J. L. Larus, *Abstract execution: A technique for efficiently tracing programs*, Software: Practice and Experience, 20(12), 1241–1258, 1990.

[3] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, N. B. Sam, *The VPC Trace-Compression Algorithms*, IEEE Trans. on Computers, 54(11), 1329–1344, 2005.

[4] M. Hron, J. Jermar, *SafeMachine malware needs love, too*, available at `https://www.virusbulletin.com/uploads/pdf/conference_slides/2014/sponsorAVAST-VB2014.pdf`, last access: July 2017.

[5] M. Zhang, R. Qiao, N. Hasabnis, R. Sekar, *A platform for secure static binary instrumentation*, Proc. of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments VEE'14, Salt Lake City, UT, USA, 28–140, 2014.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*, Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05), Chicago, IL, USA, 190–200, 2005.

[7] S. Naftaly, *Pin - A Dynamic Binary Instrumentation Tool | Intel® Software*, available at `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, 2012, last access: July 2017.

[8] M. Charney, XED Intel® X86 Encoder Decoder Software Library, available at `https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library`, 2015, last access: July 2017.