Evaluating Performance Overheads in Program execution of Scripting Languages under Virtual Machine Environment

Kumudha K N
kumudha.kn@csa.iisc.ernet.in

Shilpa Babalad
shilpab@ssl.serc.iisc.ernet.in

December 11, 2015

Abstract

Programmer productivity is gaining importance over optimized execution. Hence, many scripting languages or dynamic languages have gained popularity in the recent years. Scripting languages are widely used among statisticians and data miners for developing statistical software and data analysis. These languages are popular for their ease of use and development. They are run on virtual machines which use either an interpreter or are compiled to bytecodes, these are compiled just-in-time during execution.

The current work evaluates the overheads associated with the execution of MATLAB scripting languages under McVM virtual machine environment. More specifically, the overhead is measured in terms of instructions executed, branch prediction accuracy and cache statistics under both interpreted, JIT compiled-executed (JIT-CE) and directly JIT executed(JIT-E) modes.

We observe that compared to the interpreted mode, JIT-CE performs on an average 27 % better. Further, instruction count are 0.26 times lesser for JIT-CE and 0.21 time lesser fro JIT-E than interpreted. L1 cache miss rates reduce by 0.15 and 0.38 respectively for JIT-CE and JIT-E.

Keywords: Interpreter, JIT compiler, AOT compiler, Virtual Machine

1 Introduction

The scripting languages have gained lot of popularity in the age of big data for statistical software development and for data analysis. These languages are popular mainly because of their ease of use which in turn increases the speed of application development and productivity benefits they bring to the data analysis. They also facilitate interactive programming and easy debugging. There are more than two million users of R today and the user base is rapidly expanding [3].

The scripting languages are either interpreted or compiled to bytecodes, which are JIT-compiled at execution time under the virtual machine environment. The virtual machine is an application software program written in some programming language and is used to emulate/simulate the target computation model. It being a software program has an associated overheads with it. The present work analyses the overheads associated with the virtual machine environment in executing scripting languages. More specifically, we have considered MATLAB scripting language executing under McVM [1] virtual machine environment.

The McVM virtual machine is a component of a larger effort known as the McLab project,which was initiated by Professor Laurie Hendren of McGill university. The overall goal of the project is to find ways to improve the performance, usefulness and accessibility of current scientific programming languages. McVM is McLabs virtual machine, which implements a significant subset of the MATLAB language. It is a testing ground for new compiler optimizations aimed at scientific and dynamic languages [1, 2].

The measurements are done using VTune, commercial application for software performance analysis from Intel. We measure the overhead in terms of instructions executed, branches processed and cache statistics under both interpreted, JIT-CE and JIT-E mode.

The results give us an insight on the overheads associated with the virtual machine. This provide us an opportunity to identify the bottlenecks and overcome them in order to improve the overall execution performance.

2 Background

2.1 McVM Architecture

McVM is McLabs virtual machine, which implements a significant subset of the MATLAB language. McVM supports both interpreted and JIT compiled mode of execution. In the interpreted mode the virtual machine interprets each line of the input program and executes it immediately. Any errors cause the interpreter to halt. While in the JIT

compiled mode, the functions in the program are compiled to bytecode when it is encountered for the first time. A hash map of the currently JIT-compiled functions is maintained. Subsequent calls to the function are looked up in the hash map and on a hit the canned compiled version is directly executed. We refer to the first mode as JIT-CE (JIT compile-execute) where the program is compiled and executed. We rerun the input program while maintaining the hash map and the canned compiled binaries which we refer to as JIT-E (JIT only execute) [1, 2].

The McVM virtual machine tries to minimize performance costs incurred by MATLABs more complex and dynamic language features. This is achieved partly through the use of type-driven just- in-time specialization scheme and partly through interpreter fallback mechanism. Some costly dynamic features are still interpreted, but the JIT compiler has been designed to optimize the clearest and most common performance bottlenecks as well as possible.

Figure 1 shows the architecture of McVM virtual machine. At the core, McVMs implementation of matrix types relies directly on a set of mathematical libraries (ATLAS, BLAS and LAPACK) to implement fast matrix and vector operations (matrix multiplication, scalar multiplication, etc.). All language data types and Internal Intermediate Representation (IIR) types use the Boehm garbage collector library for garbage collection. The JIT compiler also relies on the LLVM framework to implement low-level JIT compilation to emit machine-specific code. McVM also depends on the McLab front-end, because the interpreter uses it to parse interactive-mode commands as well as source code in the form of M-files. Internally, both the interpreter and the JIT compiler rely on the language core to define the basic primitives on which they operate. This is the Internal Intermediate Representation (IIR) tree, which defines the forms valid programs can take, and the primitive data types the language supports. The JIT compiler itself depends on the interpreter because it does not emit compiled code for all operations, it sometimes uses interpreter fallback to evaluate code for which there is not yet compiler support.

The functionality of the interpreter is divided into interpretation logic and state manage- ment. The JIT compiler manages the function versioning system, emits LLVM code for the statements it can compile, and performs interpreter fallback for those it cannot. The JIT compiler also largely relies on a set of analyses to gain additional information about source programs being compiled. These analyses (live variables, reaching definitions, bounds check alimination and type inference) are crucial to generate highperformance code. McVM is entirely implemented in C++.

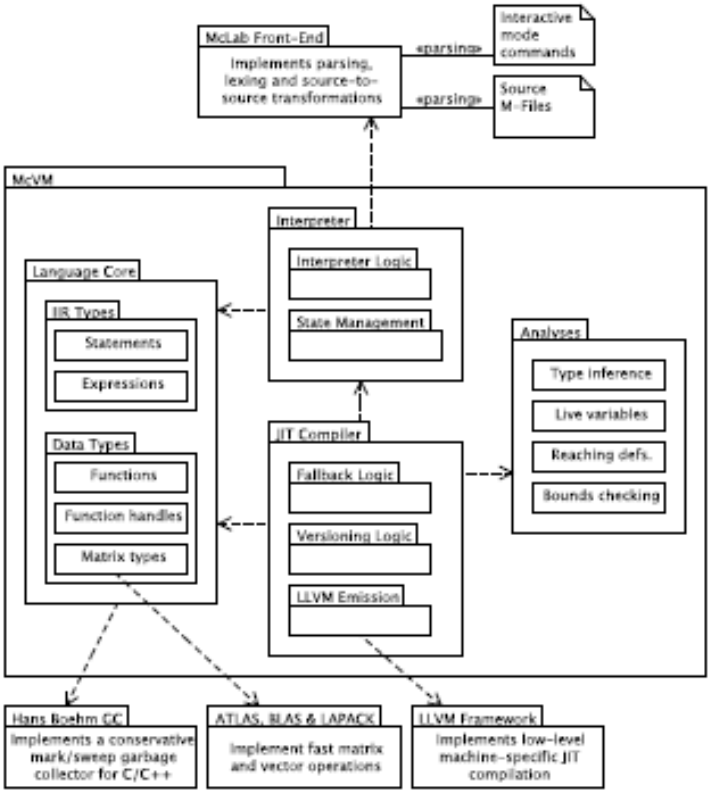


Figure 1: McVM Architecture

2.2 VTune Amplifier

Intel VTune Amplifier is a commercial application for software performance analysis for 32 and 64-bit x86 based machines, and has both GUI and command line interfaces.

The VTune analyzer has three major functions: event-based sampling, the counter monitor and call graph profiling. Event-Based sampling uses special hardware counters inside Intel MPUs to measure a number of events to monitor during the execution of a workload. Some examples of events include branch mispredictions [4].

Counter Monitor offers many different families of performance counters. They tend to be focused at the system rather than the processor level. For example, with counter monitor, it is possible to measure the number of semaphores (ensures exclusive access to data) or mutexes (ensures exclusive access to instructions) in the system Call graph profiling (CGP) uses binary instrumentation to record caller/callee relationships between functions as well as timing information. We use Event Based sampling inorder to capture our results.

Event Based Sampling mechanism is relatively simple and very low overhead. Periodically, the VTune analyzer collects data from the processor via an interrupt. The frequency that the interrupts are issued is either based on a certain number of events occurring (i.e. every 2000 instructions retired) or an external reference clock (usually the OS timer). The former mode is referred to as event-based sampling (EBS), while the latter is known as time-based sampling (TBS). Even within EBS, VTune analyzer normally calibrates itself to sample at 1ms intervals. When the interrupt occurs, VTune analyzer reads a set of registers that describe the execution context, including the dynamic execution address in memory, the associated process ID, thread ID and module. If the source code is available, then the collector can even identify the line of code that the execution address maps to.

Each experiment can take several runs of an application to actually compile all the requested data, depending on how many counters are to be collected. Unfortunately, each CPU has a fixed number of registers (4 for Woodcrest, but 12 for Montecito) used for event sampling, and these are subject to certain restrictions; some events simply cannot be measured simultaneously. VTune analyzer can gather quite a bit of data, but it does have its limits. At a 1ms (1MHz) target resolution, any workload running more than 25 minutes becomes somewhat problematic. That much data is too large to be easily manipulated and displayed; usually this is addressed by decreasing the target sampling frequency.

2.3 Experimental Methodology

We ran the experiments on a 32 core Intel Xeon machine with 128GB memory. The processor specifications are -

- L1 cache - 32KB data and instruction - 8 way set associative
- L2 cache - 256KB - 8 way set associative
- L3 cache - 2MB shared - 8 way set associative
- Branch predictor - closely resembles two-level predictor[1]

We have chosen eight benchmark programs for our study. The benchmarks and their description is listed in Table 1. From [2] we quote the functional parameters of the applications in the benchmark. The characteristics of each benchmarks in terms of the number of functions in each program, the total number of statements (in 3-address form), the maximum loop nesting depth in the entire program, and the total number of call sites are as shown in Table 2. For example, fft has two functions, 159 statements in each function, maximum loop depth of 3 and 8 call site i.e., the functions are called 8 times.

Table 1: Benchmark Description

Benchmark	Description
fft	Computes the discrete fourier transform for complex data
capr	Computes the transmission line capacitance of a coaxial pair
clos	Computes the transitive closure of a directed graph
crni	Crank-Nicholson heat equation solver
fdtd	Finite Difference Time Domain (FDTD) technique
play	Recursive minimax search
nnet	Neural network learning AND, OR, XOR functions
schr	Solves 2-D Schroedinger equation

Table 2: Benchmark Description

Benchmark	Num. functions	Num. statements	Max. loop depth	Num. call sites
fft	2	159	3	8
capr	5	214	2	10
clos	2	58	2	3
crni	3	142	2	7
fdtd	2	157	1	3
play	6	364	2	29
nnet	4	186	3	16
schr	8	203	1	32

3 Evaluation

3.0.1 Execution Time

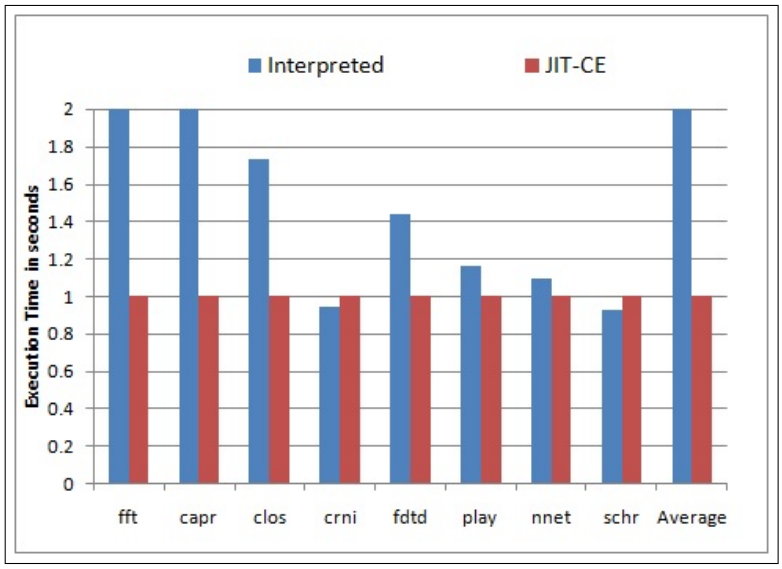


Figure 2: Execution Time

Figure 2 shows the execution time overhead of the interpreted mode over JIT-CE mode. The JIT-CE mode performs better over interpreted mode in most of the benchmarks mainly because of the com-

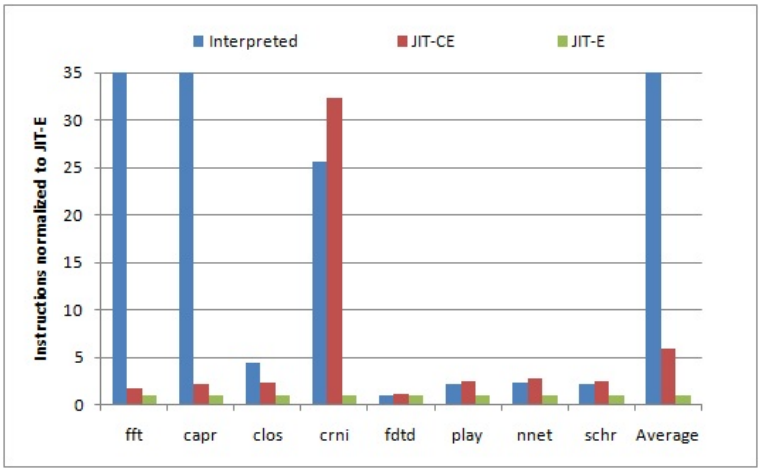


Figure 3: Instruction Count

piled version of the function calls. These functions are optimized to certain extent by the LLVM compiler infrastructure. The average sub optimality due to interpreted mode is seen to be 31 % over JIT-CE.

3.0.2 Instruction Count

The instruction count tool provided by VTune counts the instructions executed by the entire execution in both interpreted, JIT-CE and JIT-E mode. The mode of the virtual machine can be changed by setting a flag suitably. The instruction count is normalized to the JIT-E mode. The results in figure **??** show the overheads associated with interpreted mode and JIT-CE mode.

The instruction count of JIT-CE mode is less than interpreted mode in most of the benchmarks. The instruction count decreases over multiple calls to the same function in JIT-CE mode. The instructions in interpreted mode also depend on the data size on which the function is operating. For fdtd, nnet, play and schr benchmarks as the data set is very small as compared to fft and capr, the difference in performance of interpreted and JIT-CE mode is not significant. In case of crni, JIT-CE executes 32X more instructions and interpreted mode executes 26X more instructions over JIT-E mode.

3.0.3 Branch Count

The results of the branch instructions missed in all the three modes are shown in figure **??**. The results are normalized to JIT-E mode.

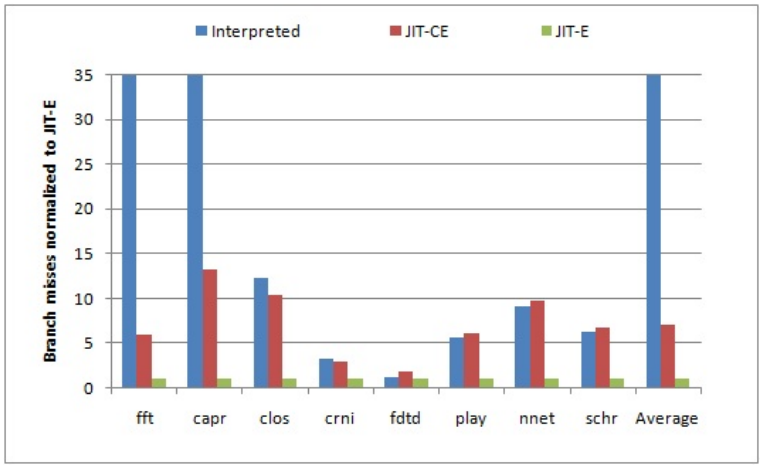
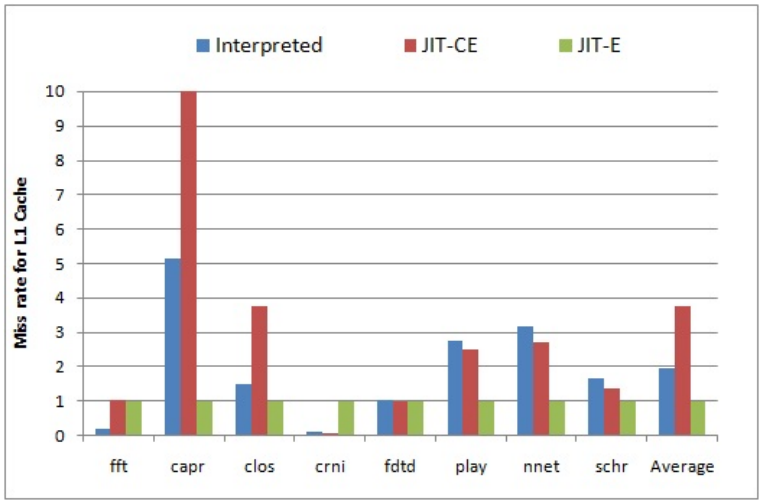


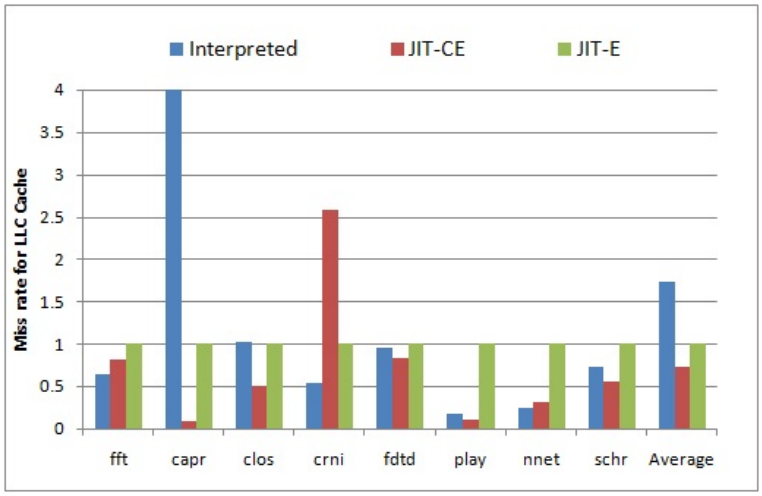
Figure 4: Missed Branch Instructions normalized to JIT-E

3.0.4 Cache Statistics

The cache statistics are collected in terms of hits and misses for L1, LLC caches for loads and stores instructions. The results are normalized over JIT-E mode as shown in figure 5a and 5b



(a) L1 Cache Miss Rate



(b) LLC Cache Miss Rate

Figure 5: Cache Statistics

4 Conclusion

It can be observed from the results presented that there is a lot of performance gap between interpreted and jit compiled version of the virtual machine. The execution time of the JIT compiled mode is better than interpreted mode by 31 %. The overhead of compiling a function and storing it for future references has cost associated with

it initially but this cost gets amortized over sequence of calls to the same function during the program execution. Hence, one has to make a judicious decision to go for interpreted or jit compiled mode which in turn depends on various parameters like number of functions in a program, instruction count, loop depth, number of calls made to the function.

Some of the optimizations done by LLVM, help reduce the overhead of interpreted language. However, the cache behavior indicates that there is significant opportunity for optimizations in JIT-compiled mode to make it more locality aware. The overheads associated with the jit compiled mode provide us an opportunity to optimize the jit compiled version further by applying rigorous compiler optimization techniques in order to achieve better performance results.

5 References

References

[1] Chevalier-Boisvert, Maxime, Laurie Hendren, and Clark Verbrugge. "Optimizing MATLAB through just-in-time specialization." Compiler Construction. Springer Berlin Heidelberg, 2010.

[2] Chevalier-Boisvert, Maxime. McVM: An optimizing virtual machine for the MATLAB programming language. Diss. McGill University, Montral, 2009.

[3] Wang, Haichuan, Peng Wu, and David Padua. "Optimizing R VM: Allocation removal and path length reduction via interpreter-level specialization." Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. ACM, 2014.

[4] Demystifying Intel Branch Predictor, Milena Milenkovic, Aleksandar Milenkovic, Jeffrey Kulick, Electrical and Computer Engineering Department, University of Alabama in Huntsville

[5] Blair-Chappell, Stephen, and Andrew Stokes. Parallel Programming with Intel Parallel Studio XE. John Wiley and Sons, 2012.

[6] Reinders, James. VTune performance analyzer essentials. Intel Press, 2005.

[7] https://software.intel.com/en-us/intel-vtune-amplifier-xe