Towards Accurate, yet Portable Performance Descriptions for Software Network Functions

Yugesh Ajit Kothari
École Polytechnique Fédérale de Lausanne
Indian Institute of Technology Kanpur
yugeshk@iitk.ac.in

ABSTRACT

The idea of Performance verification is to give any form of runtime guarantees about the performance of a software. It could be in the form of number of CPU cycles, number of cache misses, or even predicted runtime, if one can find a way to somehow predict that. Such descriptions of performance-critical software is of particular interest, and while it is difficult to develop a general framework for all softwares, exploring ways to describe performance in a specific context is beneficial for software developers and maintainers.

In the context of software network functions, Bolt[1] is a static analysis technique and tool that can predict performance of software NFs for arbitrary input workloads whether typical, exceptional or adversarial. Bolt computes these predictions in terms of Performance Contracts which are essentially functions parameterized over the input workload and environment variables that are termed as Performance Critical Variables (PVCs). One of the limitations of Bolt is that in its current form, these predictions are tied to the underlying architecture and hardware, so the predictions need to be re-computed for different platforms. In this project, we attempt to explore the possibility of predicting performance of software NFs at the LLVM-IR[2] level. These predictions (in terms of a platform-independent software level abstraction) would be portable across platforms which would make life easier for network engineers and maintainers.

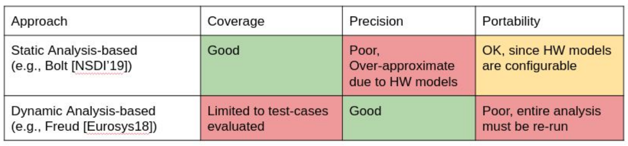
Preliminary experiments indicate that there is consistency in relative predictions at llvm-IR abstraction when compared to those in x86 (which Bolt currently works on). Since llvm-IR is not executable as-is, we believe that with some statistics and engineering heuristics, it is possible to combine the prediction abstraction of Bolt's static analysis with dynamic analysis techniques so that the end user can simply compute these contracts once, and then generate concrete performance guarantees for the NF on any platform by running just a few test cases (similar to dynamic analysis techniques for performance prediction). Our experiments show that the metrics computed are empirically similar in their trend to those computed for actual binaries in x86 and therefore suggest that LLVM can be a good abstraction to combine static and dynamic performance predictions for improved portability without significant sacrifice to accuracy.

1. INTRODUCTION

Predicting performance of softwares is an interesting problem as it gives developers good insight about how their code would behave during run-time for all kinds of inputs (normal or adversarial). One way to represent performance is by computing Performance Contracts[1]. A Performance Contract predicts performance as a function of the input workload and environment variables that together can be referred to as PCVs. These variables could be things like the number of instructions a CPU will have to execute, or the number of times the binary will touch memory, etc.

There are three pillars to a good performance prediction tool - 1) *Coverage*- how much of the input space are the predictions valid for; 2) *Precision*- How precise are the predictions when compared to the actual software in deployment; 3) *Portability*- Once these predictions are made, do they hold for different platforms that the software can be deployed on. We observe that the existing approaches are inadequate for the following reasons:

Static analysis techniques like Bolt have good code coverage as they use techniques like symbolic execution that allow us to model the behaviour of a program across all possible program



Approach	Coverage	Precision	Portability
Static Analysis-based (e.g., Bolt [NSDI'19])	Good	Poor, Over-approximate due to HW models	OK, since HW models are configurable
Dynamic Analysis-based (e.g., Freud [Eurosys18])	Limited to test-cases evaluated	Good	Poor, entire analysis must be re-run

paths. However, since static analysis techniques approximate the behaviour without actually running the program, their precision is not great. On the other hand, dynamic analysis techniques like Freud[3], which make predictions for the program paths by running it for multiple inputs, do not guarantee coverage of all program paths as that is dependent on the test suite. Since they actually run the program for the inputs test cases, their predictions are more accurate than static analysis techniques. Predictions made by dynamic analysis techniques are not portable as they need to be re-run for each different platform. Static analysis techniques offer some portability, but that is also limited to the amount of hardware configuration supported and the underlying architecture. Moreover, as the popularity of eBPF (extended Berkeley Packet Filter[4]) for managing packet processing within the Linux kernel grows, using existing techniques to make performance predictions can be inadequate, since we do not have access to the actual binary.

Therefore it is of interest to see if we can improve the portability and precision of existing techniques, and it is also interesting to see if we can somehow combine the best of both static and dynamic analysis techniques : have predictions that are reasonably precise yet portable, all the while maintaining coverage (since that is the cornerstone to fully picturing the performance of an NF).

One way to achieve this would be to compute "relative performance" across all program paths once (in the form of Performance contracts), at a level of abstraction that is independent of the underlying platform, and then generate actual predictions for a specific platform by running tests (like in dynamic analysis) and using statistical techniques to predict performance across all program paths.

If we are to follow this idea, LLVM-IR is a natural choice for the abstraction because of the following reasons : 1) *Platform independence* - LLVM-IR has a uniform abstraction across hardware and architectures (as it is a software abstraction) hence it would meet our criteria for generating portable performance predictions. 2) *Support* - Since LLVM is widely used in the industry, there is ample documentation and a wide variety of analysis tools available for LLVM (like clang[5], KLEE[6], etc). This way, the performance predictions are expressed in an intermediate human-readable representation first, that is independent of the platform; and actual predictions are concretized from these "relative predictions" based on actual runtime behaviour. Therefore, in this project, we explore the first step in using LLVM-IR for predicting performance of Software NFs - comparing the LLVM abstraction with x86 to observe if the trends in various software metrics are consistent to our expectations.

2. BOLT

Bolt is a static analysis tool that analyses NFs and generates performance contracts. Bolt assumes a clear segregation of NF code into stateful and stateless code. The stateful code is a library of data structures that the NFs use, which is written and formally verified separately[7]. The stateless code is all of the NF logic which is written by NF developers. The stateless code is verified using exhaustive symbolic execution when computing the performance contracts. Performance contracts are manually pre-computed for the stateful code, and these contracts essentially look like base case values for metrics of interest for the basic data structures (for example, value for a single PUT operation into a hashmap). Performance contracts for entire NFs (as functions over PCVs) are then computed automatically based on how the NF logic interacts with these data structures.

Bolt takes as input the NF code, replaces calls to stateful methods with corresponding calls to symbolic models and uses symbolic execution to obtain all feasible paths through the code and collects path constraints. Next it passes these path constraints to a solver to obtain concrete inputs, which when executed, will follow a particular path based on the constraints. Then, Bolt replays this concrete input and obtains a unique trace of x86 machine instructions which it traverses adding up the cost of each instruction. When it hits the call to a modelled method, Bolt uses the pre-computed contract for the stateful code appropriately. This entire process generates performance contracts for an input NF.

3. IMPLEMENTATION

We start with the existing bolt[1] infrastructure and extend it to support computation of metrics over PCVs at LLVM-IR level of abstraction. As we have already described at a high level how Bolt works, in this section we will focus on changes that were made to compute these metrics for LLVM-IR.

3.1 Instruction Traces

Since llvm bitcode is not executable as-is (unlike x86), we cannot obtain instruction traces by replaying concrete inputs. However, the fact that our Symbolic Execution[6,8] Engine (KLEE) uses LLVM, simplifies this problem for us. We modify KLEE to track all llvm instructions that it sees along a path while it is symbolically executing the stateless NF code, and once it has explored a path to completion we dump out this trace for further cost analysis. We also added command-line arguments within KLEE to specify start and end functions for tracing.

We ran into a bottleneck here which we describe as it might be interesting for the reader : LLVM provides a print API **llvm::Instruction::print** to print instructions. However, this API is only for debugging purposes and is slow. Therefore, when we tried to dump huge instruction traces, the bottleneck in our analysis became this conversion of **llvm::Instruction** to **std::string.** However, since we only require parts of the instruction trace which are relevant to our performance computation, we realised that we could cleverly avoid this bottleneck by demarcating these instructions within KLEE and printing out only the instructions we need. Moreover, we also optimised the conversion form **llvm::Instruction** to **std::string** by keeping a record of all **llvm::Instruction**(s) we already converted and therefore reduce our use of the **llvm::Instruction::print** API to once per instruction. This completes the implementation changes required to obtain the relevant instruction traces which are needed for performance computation.

3.2 Pre-computing contracts for stateful code

Similar to how the contracts were pre-computed once for the base case of data structure operations in stateful code, we must compute these contracts manually for LLVM-IR. To do this, we compile the stateful code to llvm bitcode and then analyse the bitcode to compute cost of each metric over possible paths in the stateful code. This process is semi-automated where all of the compilation and computation of each metric is done automatically if the developer of the stateful code can provide all possible program paths (in terms of a sequence of Basic Blocks in the llvm bitcode) that need to be considered for these computations. Note, that this process needs to be done only once for the stateful library of data structure operations.

3.3 Computing performance contracts

Once we have computed metrics for both the stateless and stateful code, we combine them together as one and into concrete functions over the Performance Critical Variables. Currently, Bolt supports three metrics at x86 level - Instruction count, Memory instruction count and CPU cycles. In this project we focused only on computing Instruction count and Memory instruction count for LLVM-IR as CPU cycles in its conventional meaning does not make sense for LLVM-IR(we also compare Non-Memory instructions as a consequence of these two metrics). We use simple python scripts to combine the results of Stateless and Stateful analysis together into a single file.

4. EVALUATION

We evaluate the suitability of LLVM-IR as a possible choice of abstraction by comparing the computed metric values between LLVM-IR and x86 to judge if trends are similar. We run Bolt-llvm on two NFs : NAT and Bridge and present our inferences.

4.1 Results

Figure 1, depicts the relative comparison for the combined analysis between llvm and x86 for the metric Instruction count on the NAT[9] NF (Figure 2 respectively for Bridge NF).

[1] https://github.com/bolt-perf-contracts/bolt

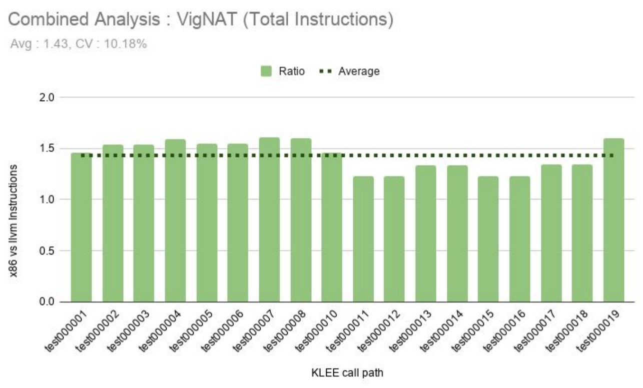


Figure 1: Comparison of Instruction conut between llvm and x86 for NAT

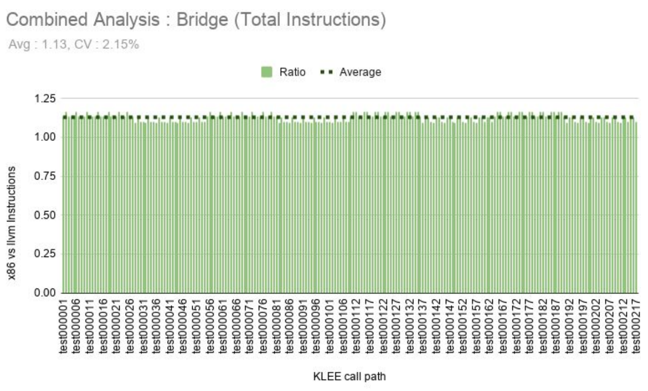
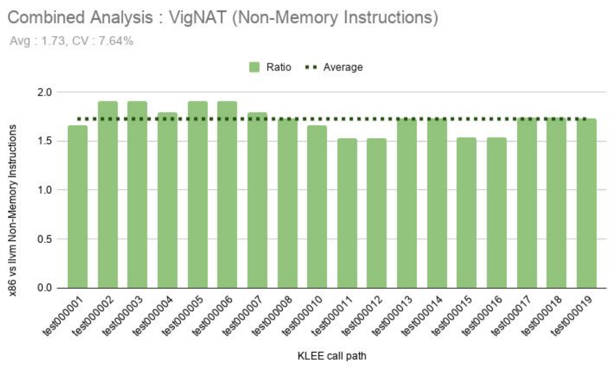


Figure 2: Comparison of Instruction conut between llvm and x86 for Bridge

On the Y-axis we plot the ratio of llvm-metric-value to x86-metric-value in order to observe how uniform this ratio remains across different call paths in the NF code. The sequence of call paths is indicated on the X-axis with no real meaning to be inferred from the order of the sequence of call paths. All subsequent plots in this section are consistent with the above description. Each plot has a dotted line that indicates the average across all call paths. Typically, we expect that there will be some degree of variation in this ratio across call paths. This is because of the difference in abstraction and expressability between llvm and x86 for example when working with memory operands, or due to the fact that llvm has an infinite register set. One aspect of using llvm would be to identify these and engineer them into the predictions to improve accuracy.

As indicated on the plots in Fig. 1 and Fig. 2, we notice CV (coefficient of variation) higher for NAT (10.18%) than for Bridge (2.15%). To investigate this, we examine individually the memory and non-memory instructions for the combined analysis. We also examine the same for the Stateless and Stateful code in both NFs separately to identify possible abstraction differences that may be causing this variation. We present our observations, inferences and only plots relevant for justifying them.



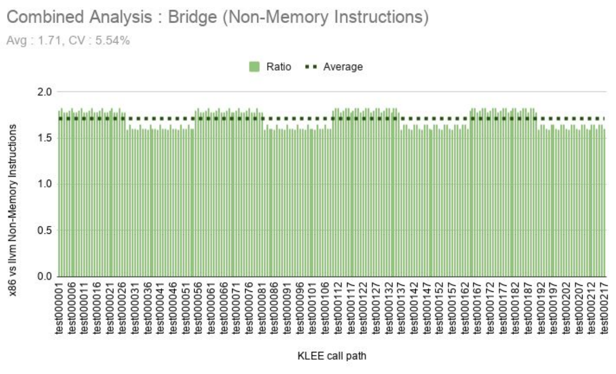
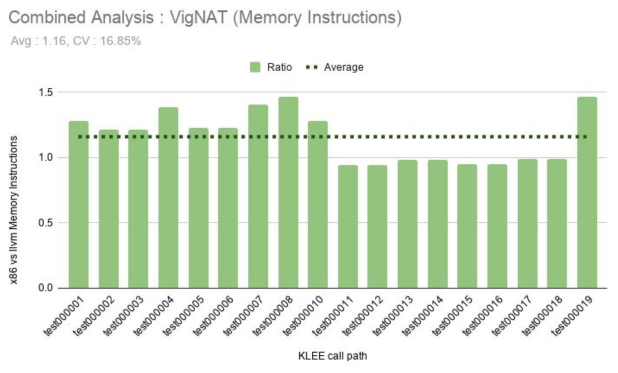


Figure 3: (Top) Comparison of Non-Memory Instruction count between llvm and x86 for NAT (Bottom) Comparison of Non-Memory Instruction conut between llvm and x86 for NAT



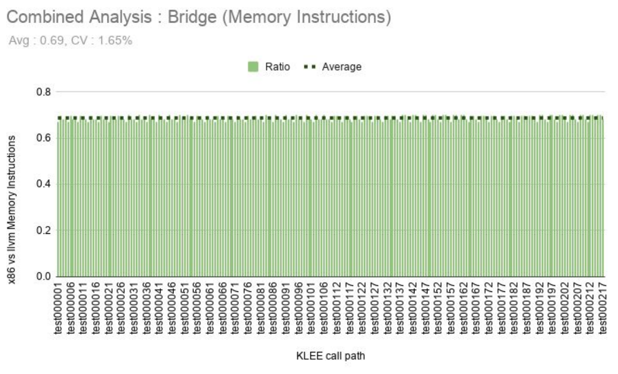


Figure 4: (Top) Comparison of Non-Memory Instruction count between llvm and x86 for NAT (Bottom) Comparison of Non-Memory Instruction conut between llvm and x86 for NAT

4.2 Observations

As stated earlier, the first observation we make is that the CV for NAT is substantially higher than that for Bridge. To understand why that could be, we observe in Figures 3, 4 values of the combined analysis for the Non-Memory and Memory instructions respectively. In the case of Non-Memory instructions we see that the CV is comparable and within a reasonable margin for both. In the case of Memory instructions, however, CV is low for Bridge (1.65%) while it is significantly higher (16.85%) for NAT. This suggests that there is some uniformity for Memory instructions across call paths in Bridge which is absent in NAT. This also suggests that the Memory instructions metric needs a finer look. We investigated this by manually analyzing the traces for Bridge and NAT. We present a comparison of Memory instructions for Stateful analysis for NAT and Bridge and also look at the outline (or pseudocode) for Bridge's implementation to draw similar conclusions empirically.

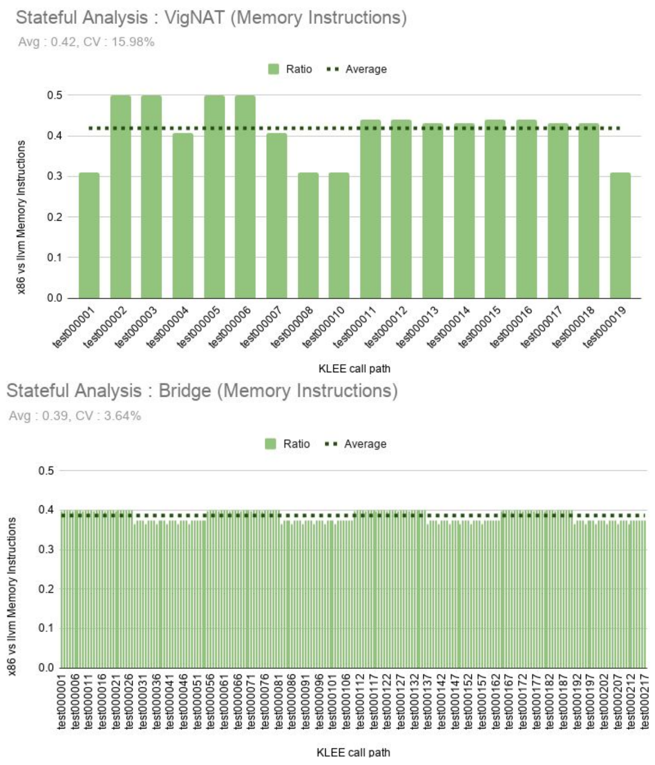


Figure 5: (Top) Comparison of Memory Instruction count between llvm and x86 for Stateful analysis of NAT (Bottom) Comparison of Memory Instruction conut between llvm and x86 for NAT for Stateful analysis of Bridge

```
from state import macTable

macTable.expireOlder(now - EXP_TIME)
macTable[pkt.src_mac] = (port, now)

if pkt.dst_mac in macTable:
  out_port = macTable[pkt.dst_mac]
  if out_port == port:
    return DROP
  else:
    return out_port, pkt
else:
  return BROADCAST, pkt
```

FIgure 6: Pseudocode for the Bridge NF

From the analysis of Memory Instructions for Stateful code, we notice that similar to the combined analysis, CV is high for NAT while it is relatively lower for Bridge. When we analysed the implementation of NAT and Bridge NFs, we realised that there are equal number of function calls in the Bridge NF across all possible call paths while this is not true for NAT. This is an important observation because function calls in x86 typically require some stack setup to allow access to both function parameters, and automatic function variables (e.g., pushing esp register, etc). This additional "overhead" for each function call is not required at the llvm abstraction.

4.3 Inferences

Based on the above observations and our investigations, we make the following inferences:

1. There is some degree of disparity between performing analysis at llvm-ir abstraction and x86 owing to overhead of function calls and corresponding stack setup and maintenance. However, it is possible to circumvent the issue by adding engineering heuristics to the analysis for llvm-ir (e.g., taking a constant overhead per function call).
2. Relative performance is mostly within 5-7% margin of error for Non-Memory Instructions.
3. Memory Instructions maintain the trend if the number of function calls across call paths is the same.

5. Summary

In previous sections we have described an initial attempt to extend the BOLT framework to generate performance predictions for software Network Functions. Currently these predictions are only done for two metrics : Total Instruction count and Memory Instruction count. For these metrics, we observe that with some added engineering heuristics it is possible to predict relative performance within a reasonable margin of error. There may be more such engineering heuristics which did not come up in the two NFs we ran experiments on, that we might need to account for. Currently, a known limitation in the implementation is that not all contracts have llvm support yet; the only ones that do are the ones we required for Bridge and NAT.

The next step in this direction would be to explore whether we can also predict relative latency (execution cycles) across call paths. Finally, we would need to explore statistical or machine learning techniques to minimize the error in predictions.

6. Conclusion

This project was merely a first step in exploring how well llvm would fare as a choice of abstraction for predicting performance of software NFs. From a preliminary evaluation and manual investigation for two NFs, it does seem possible to use llvm-ir as the choice of abstraction although there is yet a need for more exploration and experimentation to assert it with certainty. The only conclusion from this project would be that the choice of llvm does seem promising in our goal towards improving portability of performance prediction while trying to conserve accuracy.

7. Acknowledgements

I would like to thank Rishabh Iyer for supervising this project, helping, guiding and teaching me about performance verification and providing all necessary resources. Rishabh has been a most wonderful mentor and he has taught me things beyond the scope of this project, some of which shall forever help me in my future research. I would also like to thank Prof. Katerina Argyraki for giving me an opportunity to work on this project at EPFL Network Architecture Lab remotely during the COVID-19 pandemic, even though it posed many logistical challenges since it all happened remotely. Next, I would like to thank Solal Pirelli for his inputs and help in diagnosing implementation issues as and when they came up. Finally I would like to thank Prof. George Candea, and all other members of NAL and DSLAB at EPFL for the discussions they involved me in, for participating in my project presentation and posing important questions regarding the idea and the implementation.

References

[1] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance contracts for software network functions. In Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19). USENIX Association, USA, 517–530.

[2] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. Proc. of the 36th annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36), San Diego, CA, Dec. 2003.

[3] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. 2020. Analyzing system performance with probabilistic performance annotations. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 43, 1–14. DOI:https://doi.org/10.1145/3342195.3387554

[4] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. ACM Comput. Surv. 53, 1, Article 16 (May 2020), 36 pages. DOI:https://doi.org/10.1145/3371038

[5] Clang. https://clang.llvm.org/.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Symp. on Operating Sys. Design and Implem., 2008.

[7] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea; A Formally Verified NAT. In Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM'17). 2017

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In Intl. Conf. on Programming Language Design and Implem., 2005.

[9] VigNAT source code repository. https://github. com/vignat/vignat.