eBPF Optimization

Zachary Kent
Carnegie Mellon University
zkent@cs.cmu.edu

Otso Barron
Carnegie Mellon University
oebarron@cs.cmu.edu

1 Introduction

BPF, and its extension eBPF, are frameworks supporting the execution of restricted code within the kernel. To ensure safety, loading an eBPF program requires that it first passes *verification*, ensuring that the program is guaranteed to terminate safely. Verification is highly conservative, precluding dynamic memory allocation and back-edges in the control-flow graph. Bounded loops are supported through finite unrolling. Persistent state is provided through so-called "maps", which include data structures like hashtables, tries, and arrays, with size fixed at compile time. Programmers can interact with these maps, and perform other more complex functionality, by calling *helper functions*. The implementations of these helper functions are provided by eBPF and are guaranteed to be safe.

High-performance applications requiring data from kernel space are often written in eBPF to eliminate the overhead of data movement from kernel space to user space. For example, writing a load balancer in user space would require passing every packet up the networking stack from kernel to user space, and then making a decision within user space. In eBPF, this computation can be executed entirely within kernel space.

eBPF consists of a custom instruction set architecture, available as a compilation target on mainstream compilers. Once uploaded to the kernel, an eBPF program is either interpreted by an in-kernel interpreter, or Just-in-Time (JIT) compiled and executed. Given the use of eBPF for performance-critical applications, such as in the networking stack, runtime performance is of key interest.

Due to the unique restrictions of eBPF execution, running traditional compiler optimization passes on eBPF code, which were not built with these execution restrictions in mind, can generate sequences of instructions that will be denied by the verifier. The consequence is that eBPF compilation typically overlooks optimization passes that could have taken place with additional knowledge of the characteristics of the eBPF ISA and the execution environment.

Our aim is to insert eBPF-environment-aware optimizations into the LLVM stack, allowing compilation to take advantage of overlooked optimizations in the context of the eBPF ISA while passing the verifier.

Project website: https://obarroncs.github.io/15745-project/

2 Background and Related Work

The implementation of our project will largely follow the optimizations presented in [1], which presents a suite of multi-level optimizations for eBPF. These optimizations target both the LLVM IR generated by compiling the eBPF source code and the eBPF bytecode generated by translating this IR. More general optimizations, like operator-fusion, at performed at the IR level, whereas eBPF-specific optimizations, like peephole optimizations, are performed at the bytecode level.

There has been other prior work aimed at optimizing the performance of eBPF. For example, hXDP [2] compiles eBPF to a VLIW execution engine on FPGAs. The compiler performs some basic optimizations, such as translation from accumulator (2 address) instructions to 3-address code instructions and VLIW scheduling. However, many of these optimizations are ad-hoc and not applied in a principled way.

Prior work on static analysis of eBPF has been applied to, for example, implement a more precise verifier based on abstract interpretation [3]. This improves upon the existing verifier by removing the ad-hoc checks and instead presenting a technique based on rigorous mathematical foundations.

3 Project Description

3.1 Implementation

As previously stated, we plan to, for a baseline goal, implement the same optimizations detailed in Merlin [1]. This will include optimizations at both the LLVM IR and bytecode level.

(1) **Constant/Copy Propagation and DCE (Bytecode)** Compiled eBPF bytecode contains many stores of registers and constants to other registers, which can be eliminated by Constant and Copy Propagation followed by DCE.
(2) **Superword Level Merging (Bytecode)**: Superword Level Parallelism (SLP) [4] is a general technique for merging different instructions with the same opcode operating on different information into a single instruction that operates on both input registers. It was historically applied to auto-vectorizing code, but in Merlin was used to combine, for example, 4 16-bit loads into a single 64-bit load.
(3) **Macro-op fusion (IR)**: The LLVM IR generated by clang contains many instruction sequences consisting of first reading a register, performing some computation and writing it back to memory. We will fuse such sequences, where applicable, into a single read-modify-write (RMW) instruction.
(4) **Data-Alignment Optimizations (Bytecode)**: eBPF bytecode contains many loads and stores that are not word-aligned. When JIT-compiled, a single non-word-aligned operation will be compiled into multiple operations to load-/store different "slices" of the location. Properly aligning locations thus results in a lower dynamic instruction count. This is feasible for eBPF, where most loads and stores are at fixed locations (the stack).
(5) **Peephole Optimizations (Bytecode)**: Because eBPF provides only 64-bit registers, operations on integers smaller than this must use masking to ensure safety. However, this masking code can be simplifier or eliminated under many conditions, allowing for a large number of effective peephole optimizations.
(6) **Code Compaction (Bytecode)** Because eBPF operates on 64-bit registers, 32-bit loads must perform a load followed

by a mask. However, this can be simplified by instead performing a direct 32-bit load, an operation many existing compilers (including the BPF backend for eBPF) do not support.

We plan to use T-Rex to generate the network traffic used for our workloads.

3.2 Research Questions and Evaluation

We hope to answer many of the same questions that Merlin set out to address. Specifically, we hope to determine how optimization of both LLVM IR and eBPF bytecode affects static code size and dynamic performance of eBPF programs. We choose to scope our evaluation to XDP (eXpress Data Path) programs, which are a subclass of eBPF programs used to make forwarding decisions on incoming network traffic. Merlin evaluations their system on 19 XDP programs, which we will also use for our evaluation. Concretely, we wish to answer the following questions:

(1) How do IR and bytecode optimizations affect the static code size (numbr of instructions) of XDP programs?
(2) How do IR and bytecode optimizations affect the throughput of XDP programs (measured in packets per second)?

For both of these questions, we will measure the performance impact of each of the optimizations listed above individually to determine which are most useful.

3.3 Goals

75% goal: We will implement a subset of the proposed optimizations as LLVM passes to demonstrate the performance gains of eBPF-specific optimizations. Particularly, we will implement superword level merging and macro-op fusion optimizations, which take advantage of specialized eBPF instructions. We will benchmark the optimizations, with an eye on decreased instruction count and increased overall performance.

100% goal: If we meet the prior goal, we will go on to implement 4 out of 6 of the proposed optimizations to increase the performance of the generated eBPF code.

125% goal: A stretch goal is to investigate common patterns in programs targeting eBPF and implement an MILR dialect for eBPF. The optimizations in this project require domain knowledge of the eBPF execution environment to correctly transform the code. Creating an MILR dialect to encode common patterns in eBPF programs to target for specialized optimizations could allow simpler and more accurate optimizations. This would involve finding common patterns in existing eBPF programs to map to MILR constructs.

4 Logistics

4.1 Plan of Attack and Schedule

Week-by-week plan

(1) Otso: Investigate LLVM idioms for transforming code at the compiler backend.
Zak: Collect examples of eBPF code to illustrate cases we can transform.
(2) Otso: Create the code structure for a LLVM MachineFunctionPass which can modify the eBPF-specific bytecode. This will allow us to directly influence instruction selection.
Zak: Investigate the eBPF ISA and execution environment to understand what type of output instructions are desired during transformation.
(3) Otso: Begin implementing constant and copy propagation - during this, we expect to run into issues with the verifier rejecting the output.
Zak: Research more instruction sequence constraints that will guide us in future optimizations.
(4) Both team members: Being implementing superword level merging and macro-op function to meet the 75% milestone
(5) Both team members: Run preliminary benchmarks to confirm lower instructions counts in micro benchmarks. Continue validating and adding cases for SLP and macro-op fusion.
(6) Otso: Continue implementing remaining optimization passes to remove unnecessary work from programs.
Zak: Being running network-traffic benchmarks to see overall performance gains on networked eBPF programs. Collect data for final report.
(7) Both team members: Finish the final report, collecting needed benchmarks, graphs, and code snippets.

4.2 Milestone

We will set a fairly modest milestone, as our bandwidth will be quite low until the PLDI deadline passes. By Thursday, November 20th, we plan to have implemented Constant/Copy Propagation and DCE. This will require first understanding LLVM's representation of eBPF and deciding whether it is worthwhile to use this representation, or create our own. We will then have to adapt our dataflow framework to work on whichever representation we choose, and implement these optimizations.

4.3 Literature Search

See Section 2 for related and prior work. We believe that we are already fairly thoroughly versed in relevant literature, as Zak has worked in this area before.

4.4 Resources

We already have all of the resources we need to complete our project. We plan to run our experiments on Zak's advisor's machine with 4x Intel(R) Xeon(R) E7-8867, generating network traffic using T-Rex.

4.5 Getting Started

We have not done any work thus far besides writing the proposal, but do not foresee any difficulties getting started.

References

[1] J. Mao, H. Ding, J. Zhai, and S. Ma, "Merlin: Multi-tier optimization of ebpf code for performance and compactness," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 639–653. [Online]. Available: https://doi.org/10.1145/3620666.3651387
[2] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hxdp: Efficient software packet processing on fpga nics," *Commun. ACM*, vol. 65, no. 8, p. 92–100, Jul. 2022. [Online]. Available: https://doi.org/10.1145/3543668
[3] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux

kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1069–1084. [Online]. Available: https://doi.org/10.1145/3314221.3314590

[4] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 145–156. [Online]. Available: https://doi.org/10.1145/349299.349320