eBPF Optimization: Project Milestone

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

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

1 Major Changes

We have not made any major changes to the goals of scope of our project. However, based on current progress, I believe that we will likely not be able to make our 125% goal; the 100% goal already requires quite a bit of effort and should be more than enough to generate good code.

However, we have planned changing our evaluation. In particular, we no longer plan to measure realistic throughput and latency. We did this for several reasons. Although my advisor's machine does have enough NIC sockets to both send and receive packets, doing so would require unbinding at least one socket from the kernel using DPDK. If I do this incorrectly, I could accidentally unbind the NIC used for SSH connections, which would prevent anyone from communicating with it. This would be very bad, and I believe it is not worth the risk.

Moreover, we believe we can achieve an effective evaluation simply by measuring the dynamic instruction count of XDP programs under a synthetic packet workload. Although this is not a good *absolute* performance metric, it is a good relative one; we can measure the dynamic instruction count before and after our optimizations.

We plan to use the `bpf_prog_test_run`, which allows you to benchmark XDP programs under a specified workload, and then will report the total execution time. We can also record the dynamic instruction account using `perf`. Additionally, we will create an backend pass where we inject instrumentation instructions to dynamically count instructions.

2 Accomplishments So Far

So far, we have implemented Constant Propagation and Dead Code elimination and measured the performance impact of doing this using the above evaluation approach. We implemented Constant Propagation and DCE using the dataflow analyses presented in class. Although we mentioned implementing Copy Propagation in the milestone, we decided we will not do this in the end; we mistakenly included it originally, believing Merlin [1] performed copy propagation when in fact it does not. LLVM's optimizer already performs sufficient copy propagation, and there are no further opportunities for optimization.

Below is an example of constant propagation + DCE. This code transformation will occur commonly in real world programs. So far, we have seen the optimization remove instructions in real world eBPF-based packet filtering programs, namely Suricata and Katran.

```
0:    b4 02 00 00 78 00 00 00 w2 = 0x78
1:    63 21 08 00 00 00 00 00 *(u32 *)(r1 + 0x8) = w2
2:    b4 00 00 00 02 00 00 00 w0 = 0x2
```

Simplifies to:

```
0:    62 01 08 00 78 00 00 00 *(u32 *)(r1 + 0x8) = 0x78
1:    b4 00 00 00 02 00 00 00 w0 = 0x2
```

Finally, we built a test harness to upload a eBPF program to the kernel and send synthetic packets for it to handle, and measure performance using `perf`.

3 Meeting Our Milestone

We have met our milestone, modulo the fact that we decided not to implement copy propagation because it would be pointless.

4 Surprises

We have encountered several unpleasant surprises. Firstly, LLVM's target-independent backend is much more poorly documented than the middle-end, and there is also less tooling available. In particular, writing a pass on a `MachineFunction` requires adding this pass to the backend *within* LLVM, and thus we have had to fork, modify, and recompile LLVM. Merlin opts to implement these bytecode transformations in Python, presumably due to this inconvenience. As far as we know, there is no tool analogous to `opt` for backend passes.

Additionally, we have found several errors in the TableGen configuration for the BPF backend, resulting in confusing bugs within our passes. For example, the `MOV_IMM` instruction was not correctly marked as actually moving from an immediate, which broke our Copy Propagation pass. Likewise, LLVM helper functions used to detect side-effects such as "mayStore" do not function correctly due to the lack of correct instruction labeling. We fixed the TableGen configuration, and plan to upstream these changes.

5 Revised Schedule

We are somewhat behind schedule, owing to the fact that both of us have been very busy the past weeks. We will, however, have much more bandwidth in the coming two weeks and believe we can comfortably meet our 100% goal. Here is our revised schedule for the remaining two weeks:

- **Week 1**: Finish all remaining optimizations, of which there are 3. Zak will implement Superword-Level Parallelism instruction merging, and Otso will implement Macro-Op fusion and Data Alignment Optimizations.
- **Week 2**: Complete evaluation to gather dynamic instruction count for our XDP benchmarks, compiled both using our baseline and optimizations. Complete report and Poster.

6 Resources Needed

We have all the resources we need, especially since having scaled down the evaluation.

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