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.

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.

We have inserted 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.

1.1 Background

Here, we provide background information pertaining to eBPF and XDP programs.

1.1.1 eBPF Background. eBPF programs are written in C, which is then compiled down to eBPF bytecode. The eBPF ISA is essentially a restricted subset of x86 and comprises registers %r0 through %r10, where %r10 is a read-only stack pointer and the remainder are general-purpose registers. A 512-byte stack can be used to store local variables, function arguments, and spill registers.

Recall that eBPF programs must pass a verifier check before being loaded into the kernel, ensuring that they 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.

The lack of dynamic memory allocation and unbounded loops would, at a first glance, prevent the use of more complex data structures. To address this, 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.

1.1.2 XDP Background. There are many classes of eBPF programs targeted towards different applications. 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. It is common for intrusion detection systems (IDR's) and load balancers to offer compiled XDP programs to efficiently block and route packets.

An XDP program comprises one main function that receives as an argument a pointer to a *context* structure containing information about the packet to be processed, and returns a flag indicating the forwarding decision on that packet: to either drop the packet, pass it up the networking stack, or send it back to the NIC.

1.2 Prior and Related Work

The implementation of our project largely follows the optimizations presented in Merlin [1], which comprises 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 Prevail [3], a more precise verifier based on abstract interpretation. This improves upon the existing verifier by removing the ad-hoc checks and instead presenting a technique based on rigorous mathematical foundations. In fact, our optimizations improve on those in Merlin by using more principled static analyses inspired by Prevail.

1.3 Approach

We implemented several of the optimizations detailed in Merlin [1]. This includes 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 we use SLP to, 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 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).

1.4 Research Questions and Contributions

We answer many of the same questions that Merlin set out to address. Specifically, we determined 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. 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 by dynamic instruction count)?

For both of these questions, we measure the performance impact of each of the optimizations listed above individually to determine which are most useful. Answering these research questions required the following concrete technical contributions:

- An implementation of Constant Propagation and Dead Code Elimination.
- An implementation of superword parallelism. This required writing a flow-sensitive pointer analysis, dependence analysis, and scheduling algorithm.
- An implementation of Macro-op fusion
- An implementation of Data-alignment optimizations
- An evaluation of these optimizations on realistic XDP workloads.

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

2 Approach and Design Details

Here, we outline our approach to implementing the different optimizations described.

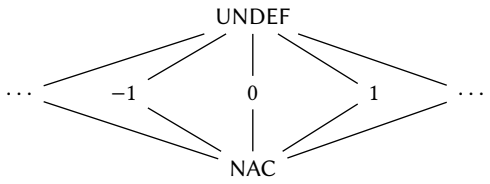

Figure 1: Constant Propagation Lattice

2.1 Constant Propagation and DCE

These optimizations were fairly simple, following the algorithms presented in lecture. We therefore only describe their implementation briefly here.

We implemented constant propagation as a dataflow analysis, associating every register with its constant value, if known, or NAC if the register does not store a constant. The lattice is shown in Figure 1. After performing this analysis, we then replace the use of every register holding a constant value with that constant. The defs of these registers holding constant values then become dead, and are eliminated by dead code elimination.

We implemented dead code elimination first by performing a live variable analysis and then deleting all instructions that define dead variables and have no side effects.

2.2 Superword-Level Merging

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 we use the approach here to pack memory operations on subword-sized locations together.

eBPF supports memory operations on locations of different sizes. Specifically, eBPF supports loads and stores to a single byte, a half-word (2 bytes), a word (4 bytes), and a double-word (8 bytes). This reflects the word size of the eBPF virtual machine, not necessarily the machine word size of the underlying architecture (which is likely 8 bytes). Thus, it is possible, for example, to pack two 4-byte stores to adjacent locations into a single 8-byte store.

We use SLP to do exactly this. Specifically, we pack independent stores of immediate (constant) values to adjacent memory locations. Following Merlin, [1], we do not use the full power of SLP to pack other memory operations, leaving this for future work. However, packing even immediate stores in a robust manner is challenging.

Our approach can be divided into several discrete phases, following roughly from [4]:

- **Adjacent Memory Identification**: We first identify references to adjacent memory locations within the source program. These are candidates for packing.
- **Dependence Analysis**: We then perform a local (conservative) dependence analysis to determine dependences between different instructions.
- **Packing**: We then pack adjacent, independent, immediate stores together into a single larger store.

- **Scheduling**: We then schedule the final instructions (some of which may have been packed) using our dependence analysis.

Each of these phases required substantial work, which we detail in the following subsections.

2.2.1 Adjacent Memory Identification. In order to identify adjacent memory references, we developed a flow-sensitive pointer analysis. Because eBPF prohibits dynamic memory allocation, this was quite tractable. In fact, almost all dynamic memory locations in eBPF exist within several fixed, statically-known *regions*:

- **Stack**: XDP programs have access to a 512-byte stack via a readonly frame-pointer, %r10.
- **Context**: The only argument to an XDP program is a pointer to the *context*, a struct containing pointers to the beginning and end of the packet along with other metadata.
- **Packet**: The context struct gives access to two pointers, pkt_begin and pkt_end pointing to the beginning and end of the packet, respectively.
- **Global**: This is the only region which is not statically known. eBPF programs can gain access to locations shared between different program instances and user space via map lookups and global variables. For simplicity, we group all of these locations into a single global region.

For our analysis, we abstract every location as a pair (*region*, *off*) consisting of a region and an offset into that region. For example, the abstraction location (Stack, 256) represents the base of the stack offset by 256. The offset can also be $\bot$, meaning that the location may fall anywhere within the region.

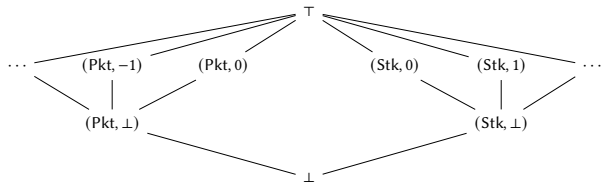

Figure 2: Selected components of the lattice used for pointer analysis

We implement our analysis as a dataflow analysis with a finite-height lattice. Using abstract interpretation and an infinite height lattice would theoretically allow for a more precise analysis, but in practice most memory accesses in XDP follow such a simple pattern that it does not make a difference. Selected components of this lattice are displayed in Figure 2. Equations governing the meet operator not implied by idempotence, associativity, or other lattice requirements are shown below:

$$(R, i) \sqcap (R', j) = \begin{cases} \bot & \text{if } R \neq R' \\ (R, \bot) & \text{if } R = R' \wedge i \neq j \\ (R, i) & \text{if } R = R' \wedge i = j \end{cases}$$

The greatest lower bound of two locations from different regions is $\bot$. The greatest lower bound of two locations residing the same region R, but with different offsets into that region, is $(R, \bot)$. This conveys the information that merging this information results in a location definitely within R but with an unknown offset.

Concretely, the domain of the analysis comprises mappings from registers to states of the lattice described above with meet and ordering given pointwise.

We now describe the transfer function $F[S](\Phi)$ parameterized by the statement S and the input mapping Φ.

$$F[x \leftarrow y](\Phi) \triangleq \Phi\{\Phi(y)/x\}$$

$$F[x \leftarrow \mathsf{Load}(loc)](\Phi) \triangleq \begin{cases} \Phi\{(\mathsf{Pkt}, 0)/x\} & \text{if } \Phi(loc) = (\mathsf{Ctx}, 0) \\ \Phi\{\bot/x\} & \text{otherwise} \end{cases}$$

$$F[x \leftarrow x + C](\Phi) \triangleq \begin{cases} \Phi\{(R, i + C)/x\} & \text{if } \Phi(x) = (R, i) \\ \Phi\{\bot/x\} & \text{otherwise} \end{cases}$$

$$F[x \leftarrow E](\Phi) \triangleq \Phi\{\bot/x\} \quad (E \text{ some other computation})$$

The first case considers the action of the transfer function at a copy assignment and has the expected semantics. The third case considers adding a constant C to some register. If before execution of the statement x stores a location (R, i), then this offset is incremented by C so that x stores $(R, i + C)$.

The second case deserves special attention. The beginning of the context region stores a pointer to the beginning of the packet, and hence a load from the beginning of the context region fetches a pointer to the beginning of the packet region. The eBPF verifier prevents stores to the context struct, so the beginning of the context will always point to the beginning of the packet at every program point.

With the machinery in hand, we can now determine whether two memory references at different program points are adjacent. For simplicitly, we represent memory references as pairs (*base*, *size*) consisting of a register *base* containing the base address and the size, where $size \in \{1, 2, 4, 8\}$ is the number of bytes accessed starting from the base location.

Specifically, to determine whether references $(r_i, size_i)$ and $(r_j, size_j)$ store adjacent locations at program points p_i and p_j, we first lookup the abstraction locations ℓ_i and ℓ_j stored by r_i and r_j at points p_i and p_j, respectively.

The references are definitely adjacent if one falls immediately after another within the same region, and thus one of the following holds:

- $\ell_i = (R, off)$ and $\ell_j = (R, off + size_i)$, or
- $\ell_j = (R, off)$ and $\ell_i = (R, off + size_j)$

In the first case, reference i falls immediately before reference j, and in the second case, reference j falls immediately before reference i.

2.2.2 Dependence Analysis. With alias analysis in hand, it becomes simple to implement a local dependence analysis. Consider a basic block comprising instructions $I_0, \ldots, I_n$ in that order. In general, I_j *precedes* I_k (or I_k depends on I_j) iff $j < k$ (meaning that I_j appears before I_k in the basic block) and I_j *conflicts* with I_k. I_j conflicts with I_k if one of the following is true:

(1) I_j writes a data item that I_k reads (RAW dependency)
(2) I_j reads a data item that I_j writes (WAR dependency)
(3) I_j writes a data item that I_k writes (WAW dependency)

We write $I_j <_p I_k$ to mean that I_j precedes I_k. The last two dependencies are technically false dependencies that could be eliminated through renaming, but this is difficult because the analysis operates post register-allocation. Further, the "data items" described in these cases can be either memory locations or registers. Detecting dependences arising from registers is simple, whereas detecting dependences arising from memory locations leverages our alias analysis and is more complex.

In particular, two memory operations at program points p_i and p_j may *conflict* if at least one is a store and the locations they access *overlap*, meaning that the ranges of bytes both operations access intersect. We determine whether two memory references overlap using our alias analysis.

In particular, we detect whether two memory references $(r_i, size_i)$ at program point p_i and $(r_j, size_j)$ at program point p_j may overlap by first determining the abstract locations ℓ_i and ℓ_j stored in r_i and r_j at points p_i and p_j, respectively. If $\ell_i = \bot$ or $\ell_j = \bot$, then conservatively we determine that the references overlap. Otherwise, it must be the case that both locations fall in a known region. If these regions are different, the references definitely do not overlap. If they are the same, we then inspect the offsets. If one is $\bot$, this means the memory reference may fall anywhere within the region, and conservatively we determine they may overlap. Otherwise, both locations $\ell_i = (R, off_i)$ and $\ell_j = (R, off_j)$ are known offsets into the same known region, so we can make a precise determination. The reference $(r_i, size_i)$ corresponds to bytes in the range $[off_i, off_i + size_i]$ and the reference $(r_j, size_k)$ corresponds to bytes in the range $[off_j, off_j + size_j]$. Thus, determining whether the references overlap reduces to checking whether these ranges of bytes do. This analysis is summarized by the predicate $overlap(\ell_i, \ell_j, size_i, size_j)$ as follows:

$$overlap(\bot, \ell_j, size_i, size_j) = overlap(\ell_i, \bot, size_i, size_j) \triangleq \mathsf{True}$$

$$overlap((R_i, \_), (R_j, \_), size_i, size_j) \triangleq \mathsf{False} \quad (R_i \neq R_j)$$

$$overlap((R, \bot), (R, \_), size_i, size_j) \triangleq \mathsf{True}$$

$$overlap((R, \_), (R, \bot), size_i, size_j) \triangleq \mathsf{True}$$

$$overlap((R, off_i), (R, off_j), size_i, size_j) \triangleq [off_i, off_i + size_i] \cap [off_j, off_j + size_j] \neq \emptyset$$

This allows us to construct the (direct) precedence relation $<_p$ and in turn its transitive closure $<_p^+$, which captures "dependence chains".

2.2.3 Packing. Now able to identify adjacent memory references and dependences, we can pack multiple memory operations into a single instruction. To do this, we first place every instruction in its own singleton pack. Then, we (iteratively) merge independent, adjacent, immediate stores of the same size until no more packs can be merged. In the base case, this entails merging two singleton packs containing stores of immediates to adjacent locations. They are then merged into a larger store-immediate which may be itself merged in a future iteration. We only merge packs that are independent, meaning that all of the instructions in both packs are pairwise independent—that is, not related by the precedence relation $<_p^+$. This ensures that the packed instructions are safe to execute in parallel. This algorithm is described in Figure 3;

```
function Pack(B = I_1, ..., I_n)
    packs ← {{I_i} | i = 1 ... n}          ▷ Initialize singleton packs
    repeat
        for all (p_1, p_2) ∈ packs × packs do
            ▷ Try to merge all pairs of current packs ◁
            if p_1 and p_2 adjacent immediate stores then
                if ∀I_1 ∈ p_1, I_2 ∈ p_2 : I_1 ≮_p^+ I_2 ∧ I_2 ≮_p^+ I_1 then
                    ▷ Packs independent ◁
                    Merge p_1 and p_2 together
    until no change
    return packs
```

Figure 3: Packing Algorithm

2.2.4 Scheduling. We can now schedule the final, packed instructions. We do so essentially by performing a lazy topological sort on the dependence graph induced by the direct precedence relation $<_p$.

Concretely, we first create a worklist containing nodes that may be ready to be scheduled. We initialize the worklist to the roots of the dependence graph; those packs with no dependencies. We then iteratively pop a pack from this worklist and check whether all of its dependencies have been scheduled. If so, we schedule that instruction as the next to be executed and add all of that pack's dependents to the worklist. We iterate until the worklist is empty and thus all packs have been scheduled. Because we never create packs with cyclic dependencies, the dependence graph is acyclic and scheduling is guaranteed to terminate successfully. This algorithm is shown in Figure 4.

```
function Schedule(packs)
    ▷ Find packs with no dependencies ◁
    ToSchedule ← {p ∈ packs | ∄p' : p' <_p p}
    while ToSchedule ≠ ∅ do
        p ← pop(ToSchedule)
        if all dependencies of p scheduled then
            Schedule p
            Add all dependents of p to ToSchedule
```

Figure 4: Scheduling Algorithm

This completes the description of the algorithm for superword-level merging.

2.3 Macro Op Fusion

Macro op fusion involves combining multiple instructions into a singular, more powerful "macro" instruction. Read-modify-write

(RMW) instructions are an example of a macro instruction - they load from memory, execute a mathematical operation, and load the resulting value back to the same memory address as the load. Modern CPU's have hardware support for RMW instruction to support atomic read-execute-write operations. In this optimization, which acts at the LLVM IR level, we identify candidate instruction sequences that can be combined into a singular atomic RMW instruction, which eBPF has dedicated instructons for.

eBPF has support for four RMW instructions: ADD, OR, AND, and XOR operations that operate with one operand as a memory location. Our pass searches for patterns of code that read from a memory location, execute one of these four binary operations, and stores the result back to memory. In the programs we evaluated, this pattern of instructions occurred commonly. After finding a matching sequence of three instruction, we swap the set of instructions with a single corresponding atomic RMW instruction. This means that every successful transformation removes two instructions from the generated eBPF code. Like other optimizations that remove instructions, this reduces load on the kernel verifier, speeds up the interpreter loop, and results in less JIT-generated instructions.

2.4 Data Alignment

Data alignment analysis provides the compiler with information on the alignment of objects in memory, providing it with the details necessary to generate safe memory access instructions. If the alignment of an object matches the memory word size of a machine instruction, the compiler can safely output a single instruction to access the memory location without the risk of runtime memory faults. On the other hand, if a compiler assumes that an object is misaligned, to avoid memory faults it may be required to break up the memory access into multiple sub-word loads or stores to construct the target value. In this optimization, we promote memory access alignments to higher, but still safe, values using knowledge of the alignment of BPF map lookups and by running a pass to propagate pointer alignment information between dependent instructions.

This optimization occurs at the LLVM IR level. A handful of LLVM IR instructions are capable of creating pointers to objects, such as alloca, getelementptr, pointer casts, bit casts, and phi nodes. These pointers contain alignment information, which the instruction selection pass uses to create machine instructions. For example, if the alignment of a pointer in a memory read is smaller than the word size, the generated machine code may consist of multiple smaller loads that are stitched together with bit shifts to construct the final value in the full word-size register.

Take the LLVM IR and the resulting generated BPF bytecode detailed in Figure 5. The code in red indicates the generated LLVM IR and BPF code without optimizations, and the code in green is generated with data alignment optimization. Without the data alignment pass, it is assumed that the pointer with the label "h_proto5" has an alignment of 1 byte, signaled by the **align 1**. The **load i16** notes that the target object is two-bytes wide. However, because the pointer is aligned to a 1-byte boundary, the compiler generates two single-byte load instructions and use a bit shift and a bitwise or to assemble the target value into the **w2** register. This single load instruction generates 4 BPF instructions.

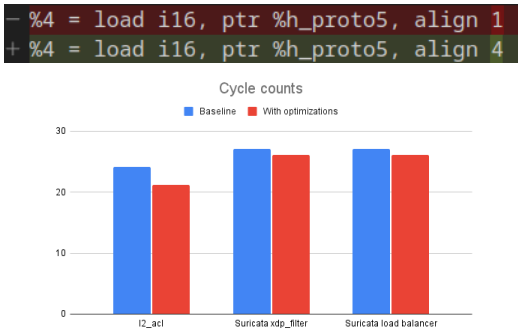

Figure 5: Data alignment optimization diff

The data alignment pass determines that the "h_proto5" pointer is aligned to a 4-byte boundary. By promoting the alignment to a higher value, the instruction selection pass doesn't need to break the load into two smaller loads, increasing the performance of the load. The effective behavior is maintained while removing three eBPF instructions. Lowering the instruction count increases runtime performance.

The return value of map lookups in eBPF are aligned to an 8-byte boundary. With this insight, the pass promotes the return value of map lookups to 8 bytes, and propagates this information through the subsequent instructions. In a final pass through the instructions, all loads and stores are examined, and the alignment of the operation is promoted to that of the target pointer. The pass uses a worklist algorithm to iterate through the instructions and propagate pointer information across dependent instructions. Our implementation improves upon the implementation in Merlin, handling phi-nodes and loops correctly to re-examine instructions where dependent pointers have new information.

3 Evaluation

We evaluate our optimizations on both synthetic, toy examples and realistic XDP programs. As mentioned, we are concerned with the following two metrics:

(1) **Static Instruction Count**: The number of instructions physically present within the bytecode.
(2) **Dynamic Instruction Count**: The number of instructions executed at runtime.

Note that static instruction count is a relatively good proxy for dynamic instruction count, as eBPF programs lack complex control flow and hot loops. eBPF only supports bounded loops, which are usually avoided anyways in XDP programs. Nevertheless, we also want to measure the effect of our optimizations on real packet workloads.

3.1 Applications

We draw our XDP benchmarks from a system used to accelerate eBPF programs on FPGA NICs [5]. This includes the following programs:

- **L2-ACL**: l2-acl implements a simple access-control list. It first checks if the packet is an IPv4 or IPv6 packet. IPv6 packets are dropped, and those that are neither IPv6 nor

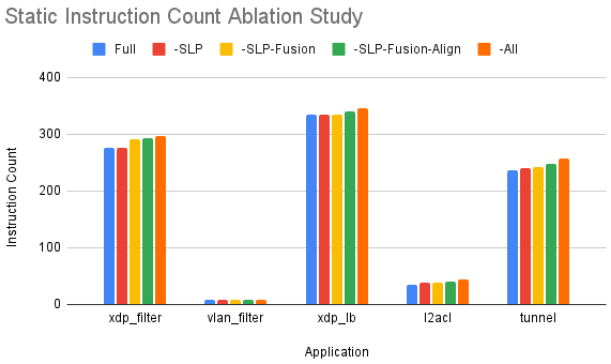

Figure 6: Ablation Study for static instruction count. "Full" corresponds to all optimizations enabled, "-SLP" corresponds to all but SLP, and so on until "-All", which is the baseline.

IPv4 are passed up the networking stack. Otherwise, given an IPv4 packet, l2-acl will check if its source MAC address is in a hashtable map. If so, the packet is passed up the stack; otherwise, it is dropped.

- **Tunnel**: Implements IP-in-IP tunneling, which encapsulates one IP-packet within another. This allows the IP packet to "travel" through a tunnel in the network between endpoints that might otherwise be unreachable.
- **Suricata**: Suricata is a general Intrusion Detection Software (IDS) comprising many different capabilities. We use the following XDP programs:
  - A load balancer that distributes incoming packets over different Suricata worker threads (**xdp_lb**).
  - A filter that drops packets from undesired flows (**xdp_filter**)
  - A filter that drops packets from certain local area network (**vlan_filter**)

3.2 Static Instruction Count

For static instruction count, we perform an ablation study, where we remove one optimization at a time and ascertain its effect on static instruction count. We first compile the selected XDP programs with clang at optimization level O2, producing current SOTA production-level code, and then run our optimization passes afterward.

Figure 6 shows the results of the ablation study. Overall, we see that our optimization suite has a measurable impact on static instruction count for the selected programs. In particular, we see that applying all of our optimizations to xdp_filter achieves an 7% speedup over baseline, and apply all of our optimizations to tunnel achieves an 8% speedup. The other programs achieve smaller yet still measurable speedups.

Across all applications, we see that data alignment optimizations have by far the largest impact, especially for xdp_filter and tunnel. This is because these applications execute many map operations. For example, xdp_filter maintains flows that should be dropped in various maps, and accesses these maps often. The lookup helper function returns a pointer to the map entry, which is guaranteed to be word-aligned on modern architectures. However, it is only byte-aligned in LLVM-IR, resulting in poor code-generation without alignment optimization; promoting these alignments results in large performance gains.

Additionally, atomic RMW operations also have a significant impact, especially on larger programs like xdp_filter and tunnel. This is because eBPF only has 10 general-purpose registers, making stack spills common in larger programs with high register pressure. Thus, RMW operations on local variables within the code are translated to RMW operations on spill slots. Translating these to atomic RMW instructions reduces the instruction count.

Unfortunately, we see that Superword-level merging, despite being by far the most complex to implement, has little impact on our examples. We hypothesize this is due to the limited scope of the merging; had we implemented full SLP and merged a larger class of instructions, we likely would have achieved larger performance gains. However, from inspecting the bytecode, clang already seems to do a good job merging memory operations, so it is unclear how many more opportunities would arise from further vectorization.

3.3 Dynamic Instruction and Cycle Count

To evaluate the runtime performance impacts of our optimizations, we measured the dynamic instruction and cycle count of various network BPF programs under network load. We built a test harness to upload XDP programs into kernel and repeatedly send realistic network traffic to the programs. The workloads are run on the same selected programs with and without our optimizations enabled, using the -02 flag. The tests were run on a machine with an Intel Xeon CPU (E7-8867), with Linux kernel version 5.15.0 running Ubuntu 20.04.6 LTS.

We selected three programs to run the benchmarks on: (1) Suricata's XDP filter, (2) Suricata's XDP Load Balancer, and (3) L2ACL. The programs are representative of a common use case for eBPF programs – efficient processing, filtering, and forwarding network packets. The behavior of these programs can be controlled through BPF maps. The Suricata filter, for example, accesses two BPF maps: "flow_table_v4" and "flow_table_v6", which contain rules to indicate the packets that the program should be inspecting. At runtime, the program will query these maps and determine if the current packet fits the description of a rule in the maps to take the appropriate action. By configuring the program in this way, we can craft network packets that cause the runtime program to execute different paths, increasing the overall coverage of instructions being executed. We use the **bpftool** to interact with maps, upload the programs to the kernel, and send networks packets to the program. We use perf to monitor the dynamic instructing count.

3.3.1 Test harness with bpftool. Our test harness utilizes bpftool to pin a BPF program, to interact with BPF maps to allow us to configure the runtime behavior, and to repeatedly run the program under a consistent workload. To simulate a realistic network load, we used pcap files captured by Warp [5] which contain comprehensive network traffic for the same programs we evaluated. Each test was run 5 times, sending 100,000,000 packets to be processed, with the average results reported.

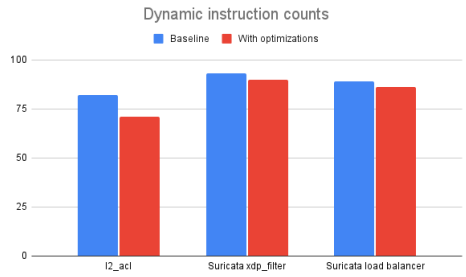

Figure 7: Dynamic instruction count benchmarks

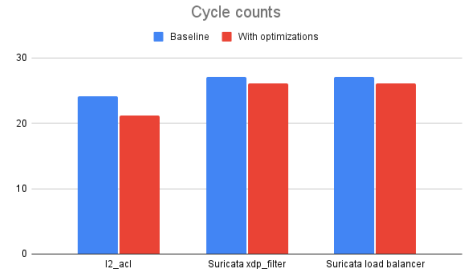

Figure 8: Dynamic cycle count benchmarks

3.3.2 Results. Figure 7 and Figure 8 report the results of dynamic instruction and cycle count for unoptimized and optimized programs. The y-axes are normalized by workload count, and contain the average number of executed instructions and cycles over the workload. The optimized programs are compiled with dead code elimination, constant propagation, SLP, macro op fusion, and data alignment optimizations enabled. All the optimized programs pass eBPF verification.

For dynamic instruction count, we observed an average decrease of 6.67% after our optimizations, and an average 6.64% decrease in runtime cycle count, demonstrating the effectiveness of the optimizations on the latest version of LLVM. These results confirm our intuitions that a decrease in static instruction count results in increased runtime performance–we see the same trends in dynamic performance metrics as we see in static metrics.

4 Lessons Learned

We learned many (hard) lessons throughout this process. Firstly, improving upon existing state-of-the-art optimizations is extremely difficult; our work amounted to, at most, an 8% speedup on eBPF bytecode already compiled by clang.

Secondly, we were reminded of the age-old adage: "premature optimization is the root of all evil". Superword-level merging was extremely complex and difficult to implement, but we found that it was not effective on the applications we selected. In contrast, low-hanging fruit like data alignment optimizations were much more effective. In hindsight, we likely should have more closely inspected the bytecode we were optimizing before deciding which optimizations to dedicate most of our time to.

5 Conclusion

We have presented a robust optimization suite for eBPF programs and demonstrated their efficacy at reducing both static and dynamic instruction count for realistic XDP programs. We believe that there are many avenues for future work. In particular, we believe that it would be interesting to explore the interplay between optimization and verification. Here, we are mostly at the mercy of the eBPF verifier, making it difficult to perform aggressive optimizations. It would instead be desirable to simultaneously perform verification *and* optimization. For example, Prevail [3]'s abstract interpretation provides all of the information needed for pointer analysis and more. The verifier itself, in general, has much more information available that allows it to perform robust optimizations. There are also more straightforward extensions of our current work. For example, superword-level merging could be extended to pack the operands to other instructions, alongside loads from memory.

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

[5] M. Bonola, G. Belocchi, A. Tulumello, M. S. Brunella, G. Siracusano, G. Bianchi, and R. Bifulco, "Faster software packet processing on FPGA NICs with eBPF program warping," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 987–1004. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/bonola