Towards Uncovering Missed Optimizations in Compilers for Programmable Packet Processing

Xin Zhe Khooi
National University of Singapore

ABSTRACT

The rise of programmable packet-processing devices has brought domain-specific languages like P4 into the spotlight, enabling the configuration of devices like programmable switches and SmartNICs. However, the reliability and performance of P4 compilers are critical for ensuring efficient hardware resource utilization and minimizing processing latencies. This paper explores potential missed optimizations in P4 compilers, focusing on enhancing performance and addressing the inefficiencies on both hardware and software targets. Drawing inspiration from similar efforts in C compilers, we propose leveraging dead code elimination (DCE) to identify these missed optimizations systematically. We introduce the concept of "optimization markers" in P4 programs to evaluate how well compilers optimize code. While the challenges posed by the diverse targets and semantics of P4 exist, we leverage the common building blocks of P4 compilers and the shared intermediate representation (IR) to enable differential testing across different compiler versions. Our proposed approach not only addresses the challenges in detecting missed optimizations but also offers a systematic way to quantify the effectiveness of P4 compilers in optimizing code. By presenting a workflow for differential testing across various P4 compiler versions, we provide a foundation for enhancing the reliability and performance of P4 compilers, ultimately contributing to the efficiency of programmable packet-processing devices.

1 INTRODUCTION

With the emergence of programmable packet-processing devices such as programmable switches and smart network interface cards (SmartNICs), domain-specific languages like P4 [2] play an increasingly important role in configuring these devices. As the prevalence of networks with such programmable devices grows, the reliability of these compilers becomes crucial. Both industry and academia have devoted various efforts [1, 10, 11] to enhance P4 compilers to ensure the correctness of these compilers.

Notwithstanding, the performance of the compiled programs is tantamount. On the one hand, inefficient compiled programs for hardware targets such as programmable switches are unacceptable. Particularly, hardware targets have a finite amount of hardware resources. Inefficient hardware utilization can result in less packet processing logic and features that can fit in the hardware [4] which can result in revenue losses for network equipment vendors and operators. On the other hand, on software targets (e.g., DPDK), inefficient compiled programs translate to additional CPU cycles spent on redundant instructions. This translates to an increase in packet processing latencies that potentially violate service-level guarantees in production networks.

In this work, we attempt to uncover these inefficiencies by identifying (potential) missed optimizations in the compilers for domain-specific languages for programmable packet processing language. To the best of our knowledge, this is the first attempt to tackle performance bugs in the P4 compiler[1]. Inspired by prior work in C compilers [12], we leverage dead code elimination (DCE) to analyze how well P4 compilers optimize code and identify missed optimizations.

We adopt a similar flow as [12] in detecting missed optimizations: *(i)* insert “optimization markers” in the basic blocks of given P4 programs, *(ii)* compute the program’s live/ dead basic blocks using the “optimization markers, and *(iii)* pinpoint the missed optimizations based on how well the compilers eliminate the dead blocks. Essentially, we exploit the fact that DCE depends on how well the prior compiler passes [3] behave, and thus one can systematically quantify how well compilers optimize code.

Adopting [12] presents several challenges despite the similarities that the C and P4 languages share: *(i)* firstly, instrumenting “optimization markers” in P4 programs is different from C which uses functions that are declared but cannot be inlined, and *(ii)* given the diverse targets (e.g., hardware, and software switches) supported by P4, one cannot simply make use of the resulting output (e.g., binary) of the compiler for analysis as in C. The semantics and instructions used can vary across different targets which results in a lack of generalizability.

We overcome the aforementioned challenges by exploiting the fact that P4 compilers, whether open-source or proprietary, are designed to share common building blocks. Fig. 1 shows the parser and the three key passes in a P4 compiler, namely the Frontend, Midend and Backend. Here, the Frontend is shared across different targets, whereas the Midend

[1]Given that P4 language is the de-facto domain-specific language for programmable packet processing, we thus focus on the P4 compiler there of.

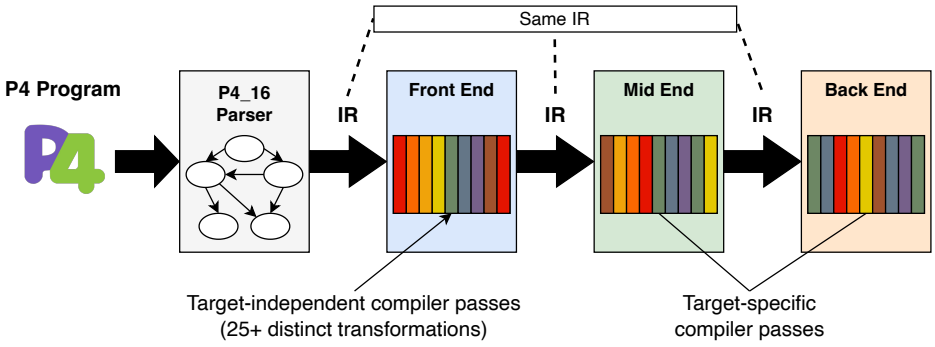

Figure 1: The P4 compiler. It consists of three key passes – Frontend, Midend, Backend.

and Backend passes are target-specific. Transformation on target-specific features, i.e., `externs`, is only done in the Midend and Backend while it is not inlined nor expanded in the Frontend. To that end, we exploit this property by declaring new `externs` as the "optimization markers" which address *(i)*. On the other hand, the same intermediate representation (IR) (which is also in P4 [3]) is used between different passes. For *(ii)*, instead of comparing the resulting output at the end of the Backend pass, we can dump the IRs before the BackEnd pass (i.e., FrontEnd and MidEnd) and search for the inserted `externs`.

The aforementioned approach allows us to conduct differential testing across different P4 compiler versions to detect potentially missed optimizations (see Fig. 2).

2 DETECTING MISSED OPTIMIZATIONS

As the P4 programs are not closed as in the C programs used in [12], it is challenging to compute the ground truth for the number of dead blocks that are actually present for evaluations. Instead, we perform differential testing across P4 compiler versions and compare their IR outputs. The overall flow is illustrated in Fig. 2.

P4 program generation: We use the random P4 program generator, Bludgeon [11], to generate the P4 programs. We check whether a generated program satisfies the following two criteria: *(i)* it contains conditional blocks, *(ii)* the program can be compiled. If the generated program does not meet the aforementioned conditions, we will re-generate another one. The generated P4 programs are named sequentially in the form of `program_XXXXX.p4`, where `XXXXX` ranges from `00000` to `99999`.

P4 Target: For generalizability, our approach should be agnostic to the P4 target. To that end, we use the `p4test` target which is designed to be primarily used for debugging the FrontEnd passes. The P4 programs generated are targeted for the `p4test`.

Instrumenting the "optimization markers": To insert the markers into the generated P4 program, we use regular expression matching to pinpoint all if-else conditional blocks and the action blocks (synonymous with methods in C). Then, we will insert markers in the form of `marker_dceXX()`, where XX here increases monotonically from 00 to 99. Once the markers are inserted, we then declare the added markers on top of the program as externs in the form of `extern void marker_dceXX()`. For the declared externs, we also annotate them with the annotation `@noSideEffects` on the advice of the P4C developers [6]. This is to hint to the compiler that these externs can be safely eliminated at the FrontEnd pass if it is within a dead code block otherwise the compiler may treat them as live code errorneously.

Compiling the instrumented programs: We download and build the Docker container images for the different versions of the P4 compiler. Using the different versions of the P4 compiler, we compile the instrumented P4 programs with the argument `--top4 End`. This dumps all the IRs in the FrontEnd and MidEnd passes which will be used for further analysis.

Analyzing the P4 IRs: We compare the `FrontEndLast` and `MidEndLast` IR of each P4 compiler version by searching for the `marker_dceXX()` patterns. Particularly, we compare the compilers by looking for markers present in one version but not the other which corresponds to (potentially) missed optimizations, i.e., differential testing. In addition, we track the number of missed optimizations across all the FrontEnd and MidEnd passes which can help us to identify root cause of the detected issues.

Implementation: We implement our approach in approximately 850 lines of Python code. The entire workflow is split into multiple Python scripts, which can be scripted. The implementation is publicly available at [5].

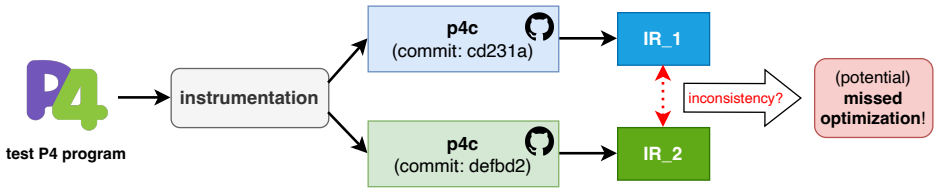

Figure 2: Detecting missed optimizations across different P4 compiler versions through differential testing.

3 ILLUSTRATIVE EXAMPLE

Here, we highlight one particular program that demonstrates how the approach enables the discovery of missed optimization opportunities in Listing 1, 2 and 3.

In Listing 1, we can observe that the program has been instrumented with 4 markers. In the `ingress` block, it invokes the action block `DwfZV` which contains dead code with `marker_dce00()`. As the computation done is irrelevant to the final packet output, this block is considered dead and should be eliminated.

We make two observations when comparing the two compiler versions, v1.2.2 and v1.2.4.5, as shown in Listings 2 and 3 based on their `FrontEndLast` IRs. In the earlier v1.2.2, we can observe that `marker_dce00()` is present while it is eliminated in v1.2.4.5. This indicates a missed optimization in the earlier v1.2.2 P4 compiler. Interestingly, even if all the markers are eliminated, e.g., `marker_dce00()`, its declaration at the top of the program is not eliminated in the later v1.2.4.5, see lines 3 to 7 of Listing 2.

We then compare the IRs of the individual compiler passes. We plot out the number of detected missed optimizations w.r.t. to the compiler pass in Fig. 3. From Fig. 3, we can observe that the first missed optimization is detected in the `SimplifyControlFlow` pass. Based on that, it helps us to narrow down the potential problematic component in v1.2.2 by looking into the code related to that pass [8], i.e., any commits between July 2021 and November 2023.

Furthermore, across versions, newly added compiler passes can also play a role. By comparing the list of FrontEnd passes in v1.2.2 and v1.2.4.5 (see Listings 4 and 5), we can see that there were additional compiler passes added before the `SimplifyControlFlow`.

Based on this gathered information, it should provide a reasonably small search space to accurately pinpoint the commit that fixed this issue, e.g., through doing git bisects. However, given time constraints on the project, we defer that as future work.

Listing 1: The program is generated using Bludgeon [11] but is redacted given the length. The action block `DwfZV` is dead as the computation done does not have any impact on the packet header fields that is eventually emitted.

```
1  #include <core.p4>
2  @noSideEffects
3  extern void marker_dce00();
4  @noSideEffects
5  extern void marker_dce01();
6  @noSideEffects
7  extern void marker_dce02();
8  @noSideEffects
9  extern void marker_dce03();
10 @noSideEffects
11 extern void marker_dce04();
12
13 // ...
14
15 action DwfZV(in ethernet_t Yklt, bit<4> emsq, bit
       <16> tSqP) {
16     marker_dce00();
17     // ...
18     ipkbiu.Zwzw = (true ? 16w37212 : (bit<16>)
       (230484601 ^ -1917033824) |-| ipkbiu.SZSs) <<
       (bit<8>)16w35785;
19     ipkbiu.SYsV = -(-1801144956 | -1545083300 -
       -735563794 & -987894174);
20     ipkbiu.SZSs = ((bit<24>)-774579209)[16:1] |-|
       16w36713;
21 }
22
23 // ...
24
25 control ingress(inout Headers h) {
26     // ...
27     apply {
28         // ...
29         DwfZV( ... );
30     }
31 }
32
33 // ...
```

4 EVALUATION

We only evaluate the publicly available open-source reference P4 compiler – p4c [7]. Commercial P4 compilers for proprietary targets are also based on this. We will be comparing six versions of the P4 compiler, i.e., v1.2.2 (July 2021),

Listing 2: `FrontEndLast` IR dumped using the P4 compiler v1.2.2 Docker image (tag: stable) from July 28, 2021.

```
#include <core.p4>

@noSideEffects extern void marker_dce00();
@noSideEffects extern void marker_dce01();
@noSideEffects extern void marker_dce02();
@noSideEffects extern void marker_dce03();
@noSideEffects extern void marker_dce04();

// ...

control ingress(inout Headers h) {
    @name("ingress.tmp") ethernet_t tmp;
    @name(".DwfZV") action DwfZV_0() {
    }
    apply {
        h.eth_hdr.eth_type = 16w9180;
        tmp.setValid();
        DwfZV_0();
    }
}

// ...
```

Listing 3: `FrontEndLast` IR dumped using the P4 compiler v1.2.4.5 Docker image (tag: 1.2.4.5) from November 2, 2023.

```
#include <core.p4>

@noSideEffects extern void marker_dce00();

// ...

control ingress(inout Headers h) {
    @name("ingress.ipkbiu") DuAzWi ipkbiu_0;
    @name("ingress.tmp") ethernet_t tmp;
    @name("ingress.Yklt") ethernet_t Yklt_0;
    @name("ingress.emsq") bit<4> emsq_0;
    @name("ingress.tSqP") bit<16> tSqP_0;
    @name(".DwfZV") action DwfZV_0() {
        Yklt_0 = tmp;
        emsq_0 = 4w2;
        tSqP_0 = 16w64905;
        marker_dce00();
        ipkbiu_0 = (DuAzWi){hQPA = 8w114,Zwzw =
     -tSqP_0 | 16w23104,SZSs = tSqP_0,SYsV = 8
    w162,xFBI = true};
        ipkbiu_0.Zwzw = 16w0;
        ipkbiu_0.SYsV = 8w26;
        ipkbiu_0.SZSs = 16w0;
    }
    apply {
        h.eth_hdr.eth_type = 16w9180;
        tmp.setValid();
        tmp = (ethernet_t){dst_addr = 48w1,
    src_addr = 48w0,eth_type = 16w15709};
        DwfZV_0();
    }
}

// ...
```

v1.2.3.0 (August 2022), v1.2.3.9 (May 2023), v1.2.4.3 (September 2023), v1.2.4.4 (October 2023) and v1.2.4.5 (November 2023). We evaluate the compilers with 10000 randomly generated P4 programs. We used an Intel Core i7-12700K-based system with 128 GB of memory running Ubuntu 22.04 for our experiments. It took approximately four hours for each individual compiler to compile the 10000 programs and dump their IRs. The tools developed for the evaluation are available at [5] while the experiment data will be made available separately[2].

4.1 Evaluation Results

Table 1 presents the total number of programs with identified missed optimizations across different versions. The table consists of six versions: 1.2.2, 1.2.3.0, 1.2.3.9, 1.2.4.3, 1.2.4.4, and 1.2.4.5. The values in the table denote the count of programs with missed optimizations detected for each combination of versions. Diagonal cells are marked with a hyphen (-) to indicate self-comparisons and other cells show the numerical count of missed optimizations between the corresponding versions. For instance, in 1.2.2, there were 0 and 258 programs that contained missed optimizations when compared with 1.2.3.0 and 1.2.3.9, respectively.

Table 1: Total number of programs with missed optimizations detected.

	1.2.2	**1.2.3.0**	**1.2.3.9**	**1.2.4.3**	**1.2.4.4**	**1.2.4.5**
1.2.2	-	0	258	258	259	259
1.2.3.0	0	-	258	258	259	259
1.2.3.9	0	0	-	0	0	0
1.2.4.3	0	0	0	-	0	0
1.2.4.4	0	0	0	0	-	0
1.2.4.5	0	0	0	0	0	-

[2]It will be provided through a separate OneDrive link via email.

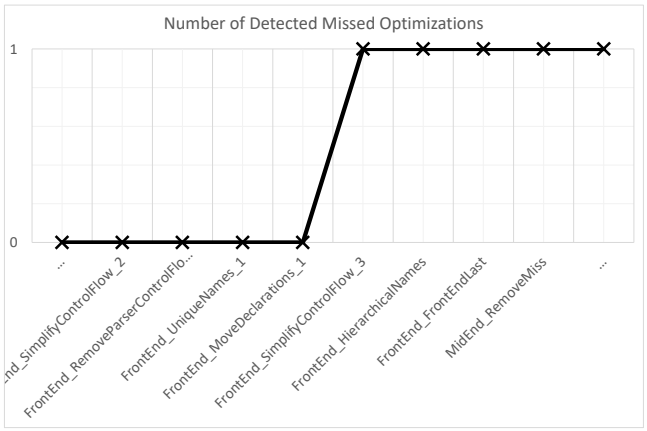

Figure 3: The number of detected missed optimizations on each compiler pass when comparing v1.2.2 to v1.2.4.5. For visualization purposes, we omit some passes as depicted in the figure with "...".

Listing 4: FrontEnd passes in v1.2.2. For brevity, we only show the last few passes from `RemoverParserControlFlow` leading to `FrontEndLast`.

```
// ...
RemoveParserControlFlow
UniqueNames
MoveDeclarations
SimplifyControlFlow
HierarchicalNames
FrontEndLast
```

Listing 5: FrontEnd passes in v1.2.4.5. For brevity, we only show the last few passes from `RemoverParserControlFlow` leading to `FrontEndLast`.

```
// ...
RemoveParserControlFlow
UniqueNames
MoveDeclarations
SimplifyDefUse
RemoveAllUnusedDeclarations
SimplifyControlFlow
HierarchicalNames
FrontEndLast
```

Following, Table 2 illustrates the total count of missed optimizations identified across various versions. The table consists of six versions: 1.2.2, 1.2.3.0, 1.2.3.9, 1.2.4.3, 1.2.4.4, and 1.2.4.5. The values within the table represent the total number of missed optimizations detected for each combination of versions. The diagonal cells are denoted by a hyphen (-) to indicate self-comparisons, while the other cells display

Table 2: Total number of missed optimizations detected across all programs.

	1.2.2	1.2.3.0	1.2.3.9	1.2.4.3	1.2.4.4	1.2.4.5
1.2.2	-	0	593	593	594	594
1.2.3.0	0	-	593	593	594	594
1.2.3.9	0	0	-	0	0	0
1.2.4.3	0	0	0	-	0	0
1.2.4.4	0	0	0	0	-	0
1.2.4.5	0	0	0	0	0	-

Table 3: Total number of programs that cannot be compiled and triggered "Compiler Bug" errors.

	compiler bugs
1.2.2	0
1.2.3.0	0
1.2.3.9	50
1.2.4.3	57
1.2.4.4	7
1.2.4.5	7

the respective counts of missed optimizations between the corresponding versions. For instance, in 1.2.2, there were a total of 0 and 593 missed optimizations when compared with 1.2.3.0 and 1.2.3.9, respectively.

Based on the findings in Table 1 and Table 2, P4 compiler version 1.2.3.9 onwards appear to have addressed bugs on earlier compiler versions, i.e., 1.2.2 and 1.2.3.0. Having said that, we noticed that not all programs can be compiled starting from version 1.2.3.9 which could have impacted the experiment figures (see Table 3). We consider the compiler bugs out of the scope of this report and defer the investigation of these compiler bugs as future work.

5 CONCLUSION AND FUTURE WORK

In this project, we have demonstrated the feasibility of leveraging prior insights into detecting missed optimizations [12] in the C compiler, specifically within compilers for programmable packet processing, such as the P4 compiler [7]. We have developed tools that assist developers in manually narrowing down the search space for "related" commits associated with the missed optimizations. While our current evaluations did not reveal any potential bugs in newer compiler versions, we have shown that this approach proves valuable when compared to older versions through differential testing. Nevertheless, this does not negate the possibility that it could unveil potential compiler bugs in later versions. At the very least, we believe this approach could serve as a beneficial addition to the P4 compiler's CI/CD pipeline,

acting as a "sanity check" to identify potential regressions alongside existing toolsets.

As part of our future work, we envision an automated pipeline that analyzes evaluation results, classifies programs, and conducts git bisects to identify problematic commits. Additionally, any detected compiler bugs should be reported automatically. Finally, since the current random P4 program generator only covers specific program blocks, we find it imperative to extend its capabilities for broader code coverage of the P4 compiler. The development of a P4 equivalent to CReduce [9] is also deemed necessary.

REFERENCES

[1] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. 2021. P4Fuzz: Compiler Fuzzer ForDependable Programmable Dataplanes. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*. 16–25.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95.

[3] Mihai Budiu. 2021. P4_16 reference compiler implementation architecture. https://github.com/p4lang/p4c/blob/main/docs/compiler-design.pdf [Accessed: Oct 2023].

[4] Xin Zhe Khooi, Levente Csikor, Jialin Li, Min Suk Kang, and Dinil Mon Divakaran. 2021. Revisiting heavy-hitter detection on commodity programmable switches. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 79–87.

[5] khooi8913. 2023. CS6223 DCE Project. https://github.com/khooi8913/cs6223-dce-project.

[6] khooi8913. 2023. Potentially Missed Optimizations in the P4C? #4246. https://github.com/p4lang/p4c/issues/4246 [Accessed: Nov. 2023].

[7] p4lang. [n. d.]. p4c. https://github.com/p4lang/p4c.

[8] p4lang. 2023. History for frontends/p4/simplify.cpp. https://github.com/p4lang/p4c/commits/main/frontends/p4/simplify.cpp [Accessed: Nov. 2023].

[9] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*.

[10] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4-16. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 136–151.

[11] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 683–699.

[12] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.