P4-DPLL: Accelerating SAT Solving Using Switching ASICs

Jinghui Jiang[†‡], Zhenpei Huang[†‡], Qiao Xiang[†‡*], Lu Tang[†], Jiwu Shu[†]
[†]School of Informatics, Xiamen University, [‡]Tan Kah Kee Innovation Laboratory

Abstract

People have been leveraging the capabilities of programmable switches, which are programmable in the data plane and process packets at the line rate, to improve the performance of distributed systems. However, few have explored whether programmable switches can speed up problem-solving. In this paper, we take a first step to explore the feasibility and benefits of this line of research. Specifically, we select the SAT problem, one of the most fundamental problems in computer science, as a case study. Our intuition is that by exploiting the parallel lookup capability of programmable switches, we can substantially speed up the process of checking whether an assignment is a solution to a SAT problem. In particular, we base on the classical DPLL algorithm and design P4-DPLL, which consists of (1) match action tables using TCAM to quickly check assignment satisfiability and find unit variables, and (2) a stack data structure using register and SRAM to efficiently make variable search decisions in the data plane. We implement a prototype of P4-DPLL and evaluate its performance extensively. Results show that P4-DPLL improves the solving time by 101x speedup on 90% quantile of all test cases, compared with a CPU-based DPLL implementation.

CCS Concepts

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Formal methods**; • **Hardware** → **Hardware-software codesign**.

Keywords

Programmable Switches; SAT Solver

ACM Reference Format:
Jinghui Jiang, Zhenpei Huang, Qiao Xiang, Lu Tang and Jiwu Shu. 2022. P4-DPLL: Accelerating SAT Solving Using Switching ASICs . In *ACM SIGCOMM 2022 Workshop on Formal Foundations and Security of Programmable Network Infrastructures (FFSPIN '22), August 22, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3528082.3544835

1 Introduction

Programmable switches have flourished in recent years because of their powerful capability, allowing network programmers to customize the algorithms in the data plane and process packets at the line rate. People have been leveraging the capabilities of programmable switches to improve the performance of distributed systems [12, 13, 17, 21, 26, 27, 30, 31]. Great benefits from the flexibility of programmable switches have been demonstrated, such as load balancing [5, 16], consensus algorithms [28], congestion control [19], and in-network caching [13, 18, 20].

However, few have explored whether programmable switches can speed up problems with high computational complexity. For example, the computational complexity of many np-complete problems increases substantially as the problem scale increases. In this paper, we take a first step to explore the feasibility and benefits of this line of research. We pick the SAT problem as an example. It is the first problem that was proven to be NP-complete and of central importance in many fields of computer science, circuit design, complexity theory, cryptography, and artificial intelligence.

We note that the main overhead in the SAT problem is the time it takes to check whether an assignment is a solution to a SAT problem. Our intuition is that by leveraging the parallel lookup capability of programmable switches, we can convert this checking process into a table lookup process in the programmable switch. Then, we can speed up the SAT solving by taking advantage of the fact that programmable switches can perform table lookups in linear time.

Specifically, we design P4-DPLL based on the DPLL (Davis–Putnam–Logemann–Loveland) algorithm, the foundation of many modern SAT solvers [7, 8]. P4-DPLL is composed of two components: a judgment component that checks whether a variable assignment will cause a conflict or generate a unit variable, and a search component that is responsible for finding assignable variables and assigning values to variables. To speed up the judgment component, we design match action tables using TCAM to quickly check assignment satisfiability and find unit variables. To accelerate the search component, we choose not to implement it in the control plane due to the long latency of the control path. Instead, we design a stack data structure in the data plane using register and SRAM to implement the process of variable search and assignment. Through the design of these two components, we completely implement the DPLL algorithm in the data plane.

We implement a prototype of P4-DPLL on Barefoot Tofino switches and commodity servers and evaluate its performance extensively. Results show that P4-DPLL improves the solving time by 101x speedup on 90% quantile of all test cases compared with a CPU-based DPLL implementation.

2 Background

SAT Problem. The boolean satisfiability problem (SAT) is to determine whether a formula of boolean variables is satisfiable or not. A boolean variable may have one of the two values, true or false. A clause is a disjunction (OR) of variables. The formulas in the SAT problem generally refer to the conjunctive normal form formula, which is a conjunction (AND) of clauses. A formula is said

*Jinghui Jiang and Zhenpei Huang are co-primary authors. Qiao Xiang is the corresponding author.

FFSPIN '22, August 22, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9329-4/22/08…$15.00
https://doi.org/10.1145/3528082.3544835

to be satisfiable if it can be made true by assigning appropriate logic values (i.e., true, false) to its variables.

DPLL Algorithm. The basic algorithm for solving the SAT problem is the DPLL algorithm [10]. It is the base of many SAT-solving algorithms such as the CDCL algorithm [6, 10, 23]. It is a backtracking-based search algorithm for deciding the satisfiability of formulas in conjunctive normal form. The DPLL algorithm iteratively goes through two phases: a judgment phase and a search phase. The judgment phase checks whether a variable assignment will cause a conflict or not and the search phase is responsible for finding assignable variables and assigning values to variables [7, 8].

For example, for the formula $(x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2 \vee x_3)$, we first choose the variable x_1 and assign x_1 to true. Because the second clause $(\neg x_1)$ evaluates to false, we go back to the first layer and reassign the variable x1 to false. Since there is no conflict and we cannot determine the truth value of the first clause $(x_1 \vee x_2)$, we choose the second variable x_2 and assign x_2 to true. We find that there is still no conflict but the truth value of the third clause cannot be determined. Therefore, we choose the last variable x_3 and assign x_3 to true. So far, we find that the truth value of all clauses is true. So we can check that the formula $(x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2 \vee x_3)$ is satisfiable. The solving process of the algorithm is shown in Figure 1.

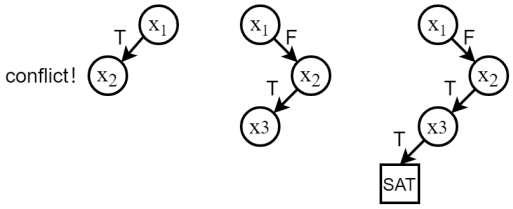


Figure 1: The solving process of DPLL algorithm.

Besides, the DPLL algorithm often uses some additional methods to reduce the solving time. A commonly used method is unit clause propagation. The basic idea is that when only one variable in a clause is unassigned and the assignment of other variables cannot make the truth value of the clause true, this clause can only be satisfied by assigning the necessary value to make this variable true. In this case, the variable to be assigned is called the unit variable, and the clause that produces the unit variable is called the unit clause. In the actual operation of the algorithm, the determination of one unit clause often leads to the appearance of another unit clause. Therefore, this method reduces a lot of search space for the SAT solving process. For example, for the formula $(x_1 \vee x_2 \vee x_4 \vee x_5) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$, we first randomly assign the variable x_1 to false. Then we apply unit clause propagation. In clause $(x_1 \vee \neg x_3)$, x3 must be assigned false making this clause satisfiable. Furthermore, in clause $(\neg x_2 \vee x_3)$, x_2 must be assigned false to make this clause satisfiable.

3 P4-DPLL Design

In this section, we describe the design of P4-DPLL that exploits programmable switches to solve SAT problems efficiently. We describe the overview of our entire design in §3.1, the design of the judgment component in §3.2, and the design of the search component in §3.3.

3.1 Overview

As illustrated in §2, the DPLL algorithm is divided into two parts: search and judgment. The search component is mainly responsible for finding suitable assignable variables in the formula and assigning them, and the judgment component is mainly responsible for judging whether the current assignment will lead to some situation.

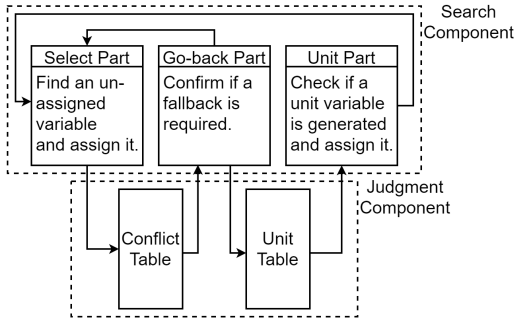


Figure 2: P4-DPLL structure.

Following this division, we propose P4-DPLL, a SAT solver that exploits the parallel lookup capability of switches for speedup, as shown in Figure 2. To avoid the overhead caused by the control path, we deploy both components in the data plane. To carry the data required by the SAT solver, we design two dedicated headers, as shown in Figure 3. The P4-DPLL fields are inside the Ethernet payload and a special EtherType is reserved for P4-DPLL. The switch uses this type to invoke the custom packet processing logic.

Figure 3: P4-DPLL packet format.

The Judgment Component. The judgment component has two functions: the first is to judge whether the current assignment will cause a conflict, and the second is to judge whether the assignment will generate a unit variable. We implement these functions on P4 using two TCAM tables. The first one we call the conflict table, which is used to determine whether there is a conflict. And the second is called the unit table, which is used to determine whether there is a unit variable. By leveraging the ability of TCAM table parallel lookups, we achieve speedups for both functions.

The Search Component. The main function of the search component is to search for assignable variables, assign values to variables, and handle special cases. A key challenge in the search component is that it needs to record the search history for backtracking. To address this challenge, we implement a stack structure on the switch using registers and SRAM tables, which is also our key design in the search component. By using this stack, we are able to record the search history on the switch and go back to the correct point when appropriate.

3.2 Judgment Component

Conflict Table. In a SAT formula, if an assignment causes a conflict, it must be because there is a clause that evaluates to false under the current assignment. For a clause of the SAT formula, evaluating to false means that all variables in the clause evaluate to false. So no matter how long a clause is, there is one and only

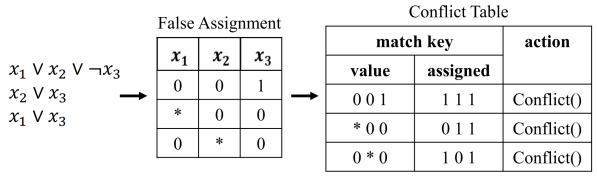


Figure 4: SAT to conflict table. The false assignment of the formula can be generated according to the SAT formula, and then the conflict table of the SAT formula can be generated according to the false assignment.

one assignment that makes the clause evaluated as false, which we call false assignment.

For example, in the formula on the left side of Figure 4, for the first clause, it is false only if assigned x_1 to false, x_2 to false, and x_3 to true. For the second clause, only if assigned x_2 to false and x_3 to false, this clause is false. We use 0 to indicate that the variable assignment is false and 1 to indicate that the assignment is true. Because the second clause does not contain x_1, the value of x_1 is recorded as *, which is arbitrary. In this way, we can generate false assignments for each clause. And so on, we can get false assignments for this formula. For any SAT formula, we can get a corresponding false assignment.

Because if a false assignment occurs in an assignment, it can be concluded that the assignment will lead to a conflict, we can construct a conflict assignment match table based on the false assignment, called a conflict table.

As shown in Figure 4, the key of the conflict table is the value and assignment of the variable, each bit represents a variable, the action is the conflict handling function, and each entry of the conflict table corresponds to the false assignment of a clause in the SAT formula. In this way, as long as the conflict table hits, there must be at least one row of entries appearing in the current assignment, and there must be at least one false assignment appearing in the current assignment. Therefore, as long as the conflict table hits, it can be asserted that the current assignment will cause a conflict so that it can be transferred to conflict processing.

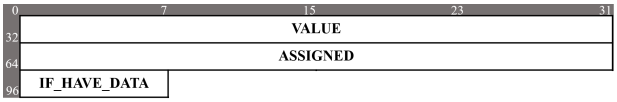


Figure 5: JUDGMENT header.

In order to carry the variable information to match the conflict table, we use the JUDGMENT header as shown in Figure 5. Limited by the computing power of the programmable switch, we set the length of the VALUE and ASSIGNED fields to 32 bits, which is exactly the upper limit of the length of the switch to perform computing operations. There are a total of 8 packet headers, which together can represent 256 variables, which is exactly the size of a conflict table.

The Formula Partitioning Algorithm. In commodity programmable switches [4], a TCAM table is no longer than 512 bits wide. Because the conflict table has two fields of equal length, value and assigned, a conflict table can process conflict situations of up to 256 variables at the same time. To handle larger SAT formulas, we design a simple formula partitioning algorithm. By dividing a large SAT formula into several small parts, and then putting the

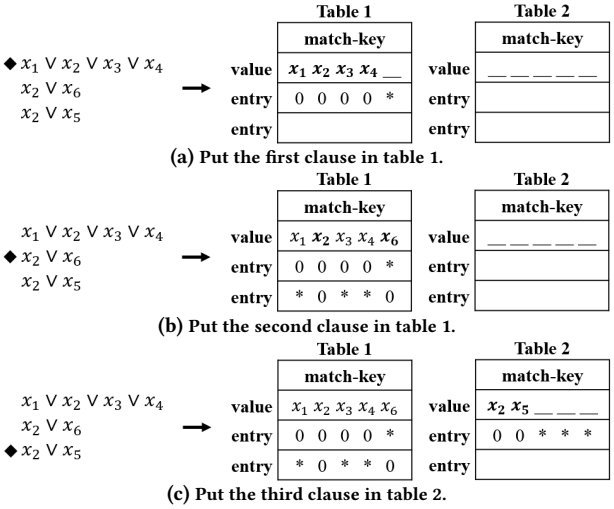


Figure 6: An illustration example to demonstrate the formula partitioning algorithm.

small parts into the corresponding conflict table one by one, the compatibility of P4-DPLL to larger formulas can be realized.

The algorithm divides the entire formula according to the width of the table's match-key. It traverses all clauses of the entire formula and then decides whether to divide the clauses into the current match table based on whether the table has enough capacity. If the table's match-key has enough free width, and the table is not full, the algorithm divides the clause into the current table. If the table's match-key has enough free width, but the current table is full, the algorithm will not divide the clause into the table; and vice versa. Of course, if all the variables contained in a clause are already contained in the table, and the table is not full, the clause will also be included in the table.

For example, suppose the match-key width of the table is 5 bits, and it can only contain at most 2 entries, as shown in Figure 6. For the first clause, since table 1 is empty at this time, it can be directly put into table 1. For the second clause, it has a variable already included in table 1, so if it is to be put into table 1, it needs to consume one-bit width on the match-key of table 1. At this time, the match-key of table 1 still has enough free bits, and table 1 is not full, so the second clause can be put into table 1. If we want to put the third clause into table 1, the situation is the same as in the second clause. But now there is not enough bit width in table 1, so only the third clause can be put into table 2.

Although this algorithm is not optimal, it does expand the capacity of our table during our evaluation. Improving this algorithm is our future work.

Unit Table. The main function of the unit table is to quickly determine whether a unit variable is generated. It is similar in principle to the conflict table. For a clause, a unit variable is created if all but one of the variables are assigned and evaluated as false. In this case, the condition for no conflict is that the remaining variable evaluates to true. That is, for a clause with k variables, there are k ways to generate unit variables.

As shown in Figure 7, for a clause with 3 variables, it generates 3 different unit assignments. As with conflict assignments, when a unit assignment appears in the current assignment, the current

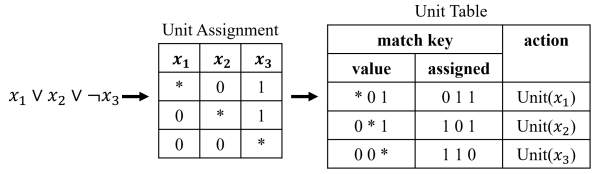


Figure 7: SAT to unit table.

assignment must generate the unit variable throughout the SAT formula. Therefore, the unit table converted from the unit assignment can achieve similar functions to the conflict table. When the unit table hits, it means that the current assignment must generate the unit variable in the entire SAT formula.

The conflict table directly returns the result and calls the conflict handling function, but the unit table returns an additional parameter, which represents which unit variable is generated by this entry. The unit table does not need to consider the priority of entry. For the unit table, all unit variables generated by the current assignment have the same priority. After processing the first unit variable, the remaining unit variables will be recognized and processed again when entering the unit table later.

3.3 Search Component

The core of the search component in P4-DPLL is a stack to track search history during SAT solving. Since the search space required to solve the SAT formula is huge, we deploy the stack on the register instead of the packet header. Based on this stack, the search component is divided into three parts: search part, go-back part, and unit part. The search and go-back parts operate the stack to complete the advance and reverse of the algorithm. The unit part does not operate the stack, it mainly implements the processing of unit variables.

Because the search component of the P4-DPLL is complex, we implement it by recirculating packets multiple times on the switch. We use the OP field to identify what the P4-DPLL should do while processing the packet, and IF_OP_DONE to indicate whether the current action is over. The control information such as the OP field of the search component is mainly placed in the SEARCH header, as shown in the Figure 8.

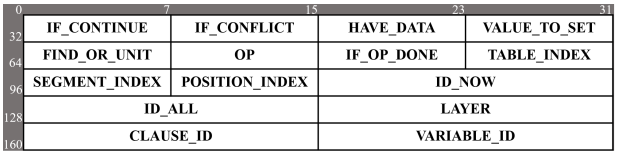


Figure 8: SEARCH header.

Figure 9 shows a schematic of the stack. We put the get and push operations of the stack on the same stage (in fact, due to atomic constraints, they can only be placed on the same stage), but the conditions required to perform these two actions are different. When OP is OP_PUSH, the match table hits and the push operation will be performed. The get operation cannot be performed at this time. When the OP is OP_GET, the match table hits and the get operation will be performed. The push operation cannot be performed at this time. If the solver needs to get the data immediately after the push operation, it needs to modify the op field and recirculate the packet back to the parser.Note that this stack only supports single-thread

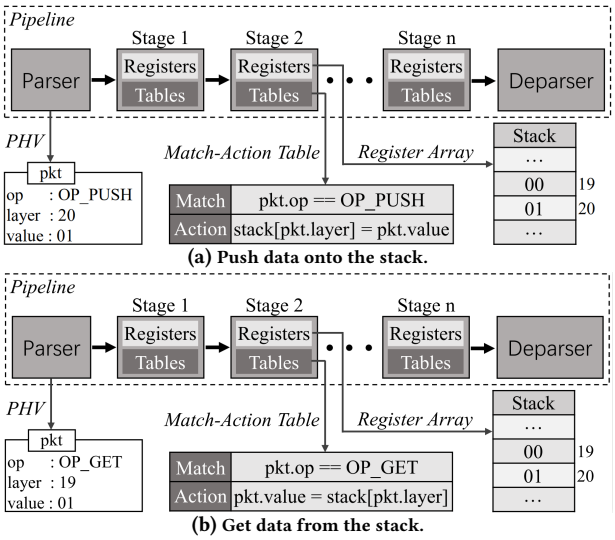


Figure 9: Pipelines in programmable switches. When op is equal to OP_PUSH, data is pushed; when op is equal to OP_GET, data is taken out.

search. Extending it to support parallel search is one of our future works.

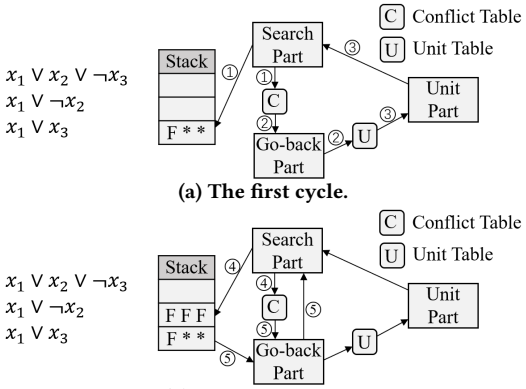
The functions of each part in the search component are relatively independent, dealing with the input and output of the conflict table and the unit table respectively.

The Search Part. In this part, P4-DPLL will linearly look through all current variables to find an assignable variable. To avoid starting a search from scratch every time, P4-DPLL maintains a search pointer and ensures that all variables preceding the pointer have been assigned. After finding an assignable variable, P4-DPLL assigns it and generates the input of the conflict table. When finished with these operations, the search part pushes the data onto the stack and passes the data to the conflict table.

The Go-back Part. After sending to the conflict table, the packet will enter the go-back part. In this part, the solver decides whether to start go-back processing or to start looking for the unit variable based on the conflict table matches. If the go-back process starts, the solver will keep backtracking on the stack and reassigning variables until there are no more conflicts. If the switch keeps going back until there are no variables to choose from, the solver will report UN-SAT. The solver will start matching the unit table If there are no conflicts.

The Unit Part. In this part, the solver will judge whether the current variable will generate a unit variable according to the results of the unit table. If so, the unit part will report this to the search part and start assigning values to the unit variable. If no unit variable is generated, P4-DPLL modifies the OP field, returns to the first part, and looks for the next variable that can be assigned a value.

Take Figure 10 as an example. In the first step, the search part searches for unassigned variables, and it finds x_1. It assigns x_1 false, pushes the result in the stack and enters it into the conflict table. In the second step, since there are no conflicts, the go-back part enters the current assignment into the unit table. After entering the unit part, since a unit variable is generated, it starts to process the unit variable, as shown in the third step.



(a) The first cycle.

(b) The second cycle.

Figure 10: An example of the search component.

In the fourth step, the search part receives the data of the unit part and pushes it onto the stack. However, since the current assignment would cause a conflict in the formula, the go-back part gets historical data from the stack in the fifth step and returns to the search part to start reassignment. A more detailed example can be found on our website [3] with a technical report [14].

4 Performance Evaluation

4.1 Setup

Prototype Implementation. We implement a P4-DPLL prototype on Barefoot Tofino switches. We also implement a simplified version of P4-DPLL called PServer which means we use both P4 and server to solve SAT problems. PServer solves the SAT problem through the interaction between the programmable switch and the server by deploying the search component of P4-DPLL on the server and deploying only the judgment component on the programmable switch. The server will first search for variables. When it needs to make a judgment, PServer will send a packet carrying the current assignment information to the programmable switch for judgment through DPDK. In this way, the programmable switch and the server are constantly interacting. Finally, the server will get the solution to the SAT problem. To better evaluate the performance of P4-DPLL, we also design a DPLL algorithm that is completely implemented on the server-side, called ServDPLL.

Note that the search components of ServDPLL and PServer are implemented slightly differently. This is because if we apply the search component in PServer to ServDPLL, it will substantially increase the judgment time of ServDPLL, and many test cases will time out. Therefore, we introduce an optimization in ServDPLL to digest the formula while assigning the value to the formula on the search component. Note that in this preliminary evaluation, we do not compare P4-DPLL with other hardware-based acceleration methods such as FPGA-based SAT solver [24, 29]. This is an ongoing future work.

Testbed. We perform experiments on a Barefoot Tofino switch [4] and a server with 2 Intel Xeon Silver 4210R CPUs, 128GB memory, and a Mellanox ConnectX-5 NIC card.

Datasets. We collect formulas from two public datasets [1, 2]. There are 39,949 formulas in these two datasets, of which 39,928 were satisfiable and 21 were unsatisfiable. The scales of the variables in these formulas range from 20 to 600, and the scales of the clauses range from 80 to 2,237.

Comparison Methods. To study the performance of our solver, we used five versions of the solver to compare: P4-DPLL, PServer, ServDPLL, Z3, and MathSAT. MathSAT and Z3 are the most commonly used solvers for solving SAT problems with good performance in the current application. Among them, MathSAT is implemented based on the DPLL algorithm but with many advanced scenario-dependent optimizations, and Z3 is implemented based on the CDCL algorithm. We choose not to compare P4-DPLL with any parallel SAT-solving algorithms because parallel SAT solver is complex and does not necessarily have good accelerations.

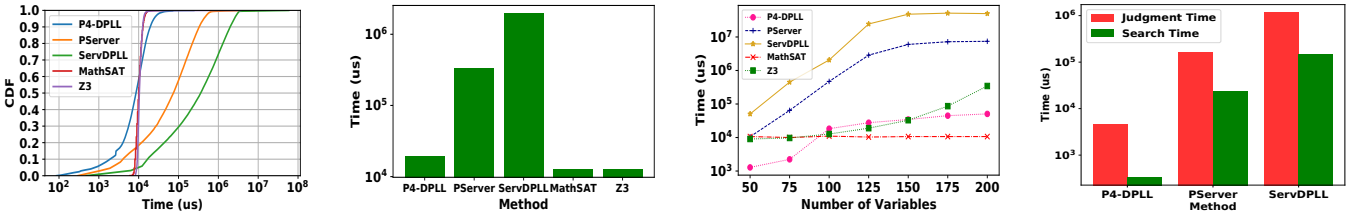
Metric. We test and analyze the solving time of different methods of SAT solvers on the same data set. To fairly compare the time-consuming of each solver, we do not include the time spent on pre-processing. To analyze the time advantage area between each method, we further divide solving time into search time and judgment time to analyze the time advantage area between each solver.

4.2 Results

Overall Performance. We perform statistics on the solving time for the formulas collected from the dataset. Figure 11a plots the CDF and Figure 11b gives the 90% quantile of all test cases for each method. We can see that P4-DPLL improves the solving time by 101x and 16x respectively on 90% quantile of all test cases compared to PServer and ServDPLL. Besides, from Figure 11a, we can see that the SAT solving time of P4-DPLL is faster than that of MathSAT and Z3 on 50% quantile of all test cases. The reason why MathSAT and Z3 have a large portion of instances that are faster than P4-DPLL is that Z3 uses CDCL which is inherently much faster than DPLL and MathSAT applies many instance-dependent optimizations on DPLL such as variable elimination, subsumed clause removal, and backwards subsumption [6]. Realizing instance-dependent optimizations in P4-DPLL is our future work, such as by recording the polarity of variables to achieve better assignments.

Performance on Different Formula Size. To analyze the impact of formulas of different sizes on the performance of the sat solver, we select and divide the formulas in the dataset into eight different categories according to the number of variables and the number of clauses. For different types of formulas, we separately analyze the evaluation results of different SAT solvers. Figure 11c plots the solving time on 90% quantile of cases on different formula sizes. With the increase in the number of variables in the formula, the solving times of PServer, ServDPLL, and Z3 increase significantly. MathSAT's solving time and P4-DPLL's solving time don't vary significantly. In addition, when the formula is small, P4-DPLL has obvious advantages in performance over the other solvers. When the formula is relatively large, P4-DPLL has a similar performance to MathSAT.

Judgment Time and Search Time. Figure 11d gives the judgment time and search time for each method. P4-DPLL solver has the smallest judgment time, and ServDPLL has the longest judgment time. This is because P4-DPLL uses the table of the programmable switch to make judgments, and the judgment result can be obtained in linear time. Although PServer also uses the programmable switch to determine whether an assignment conflicts, it needs to send and



(a) CDF of SAT solving time.

(b) SAT solving time on 90% quantile of all test cases per method.

(c) SAT solving time on 90% quantile of cases on different formula size.

(d) Judgment time and search time on 90% quantile of all test cases per method.

Figure 11: P4-DPLL Performance Evaluation.

receive packets frequently through the server in order to obtain the judgment result of the switch. So the judgment time of P4-DPLL is shorter than the judgment time of PServer. We can see that P4-DPLL improves the judgment time by 246x and 35x respectively compared to PServer and ServDPLL. Besides, we can see that P4-DPLL has the largest proportion of judgment time. This is due to the fact that P4-DPLL implements the algorithm using the stack, which requires multiple calls to the stack for each check. This greatly increases the time spent on searches, which in turn results in a reduced share of judgment time. We can also see that the search time of ServDPLL is increased compared to that of PServer. This is because we introduce an optimization in ServDPLL to make the judgment time shorter and the search time longer on ServDPLL.

Summary. In summary, P4-DPLL has a significant performance improvement compared with PServer and ServDPLL. Besides, the SAT solving time of P4-DPLL is faster than that of MathSAT and Z3 on 50% quantile of all test cases. Specifically, the P4-DPLL has a clear performance advantage for small formulas. For relatively large formulas, the P4-DPLL solver is comparable to the currently popular MathSAT solver. Besides, the judgment time of the P4-DPLL is significantly smaller than that of the PServer and the ServDPLL.

5 Discussion

In this paper, we take a first step toward using programmable switches to accelerate problem-solving. Here, we discuss some limitations of programmable switches in this direction.

Resource Constraint. The core idea of P4-DPLL is to use the quick matching capability of the TCAM table to speed up the DPLL algorithm. However, on existing programmable switches, the size of the TCAM that can be used is limited.

For a TCAM table, the maximum bit width is 512 bits, and under this condition, the maximum number of entries that can be used in a single stage is 1024. In our implementation, we use a P4 switch with 12 stages, which means that our conflict table and unit table cannot accommodate an excessive number of clauses.

To handle the formula as well as possible, we empirically allocated the TCAM resources to the conflict table and the unit table in a ratio of one to three. This allows P4-DPLL to accommodate a maximum of 768 variables or 3,096 clauses in a formula. Depending on our implementation, it is possible to implement P4-DPLL together with multiple switches or multiple pipelines of a switch, and we leave it as future work.

The design of P4-DPLL supports solving multiple SAT formulas with one data plane configuration. But, in reality, this is constrained by current P4 hardware.

Reconfiguration Constraint. A fundamental constraint of P4 is that updating the match table on the switch is a time-consuming operation. Due to this limitation, our P4-DPLL cannot make instant updates to the conflict table and the unit table at runtime. Therefore, it is difficult to extend the design of the P4-DPLL to other more efficient SAT algorithms, such as CDCL [22]. We believe this is a key obstacle to accelerating other algorithms using programmable switches. We plan to investigate this issue in-depth in the future.

Integration Constraint. P4-DPLL could be integrated into Z3 or MathSAT as the judgment part in theory. However, this is constrained by the overhead of control path. How to efficiently implement integration is our ongoing work.

Cost. One may think using programmable switches to solve the SAT problem is not financially efficient. However, we are exploring solving multiple SAT problems using programmable switches to reduce the cost further.

6 Related Work

In-network Computing. The networking community has extensively explored to improve the performance of distributed systems such as load balancing [5, 16], consensus algorithms [28], machine learning [25], key-value stores [13, 18, 20], and network telemetry [9, 11]. Few have explored whether programmable switches can speed up problems with high time complexity such as NP problems. We believe we are one of the first to explore the feasibility and benefits of this line of research.

SAT Solving. The DPLL algorithm is the basic algorithm for solving SAT problems. Many modern solvers use the CDCL algorithm, which augments the DPLL algorithm with efficient conflict analysis [7, 8, 10, 22]. Implementing the CDCL algorithm needs updating the match table on the programmable switch on the fly, which consumes lots of time. Therefore, the advantage of using programmable switches may be less significant. We leave it to future work.

Some people use FPGA to accelerate SAT solving by leveraging its increasing design capacity, high performance, massive flexibility, and parallelism [15]. Comparing the acceleration performance of FPGA-based SAT solvers and P4-DPLL is an important future work for us, which can help us better understand how to accelerate SAT problem-solving.

7 Conclusion

As a first attempt to explore whether programmable switches can speed up solving problems with high computational complexity, we design P4-DPLL to accelerate SAT-solving using P4 switches. In the future, we plan to also study how to use programmable switches to accelerate other problem-solving such as graph process.

Acknowledgments

We thank all the comments from the anonymous reviewers. We also thank Rongqiang Chen for his valuable feedback on the paper. This work is supported in part by NSFC Award #62172345, Alibaba Innovative Research Award, Open Research Projects of Zhejiang Lab #2022QA0AB05, Future Network Innovation Research Award of Ministry of Education of China #2021FNA02008 and Tan Kah Kee Innovation Laboratory Award #HRTP-2022-34.

References

[1] 2000. SATLIB - Benchmark Problems. https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.
[2] 2022. The International SAT Competition. http://www.satcompetition.org.
[3] 2022. P4-DPLL. https://p4-dpll.github.io/.
[4] Barefoot 2019. Barefoot S9180-32X Switch. https://www.ufispace.com/uploads/able/files/productfilemanager/000045467d1fc648d792c404372956a0.pdf.
[5] Cristian Hernandez Benet, Andreas J Kassler, Theophilus Benson, and Gergely Pongracz. 2018. Mp-hula: Multipath transport aware load balancing using programmable data planes. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*. 7–13.
[6] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 93–107.
[7] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.
[8] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM (JACM)* 7, 3 (1960), 201–215.
[9] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.
[10] Youssef Hamadi and Lakhdar Sais. 2018. *Handbook of Parallel Constraint Reasoning*. Springer.
[11] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 285–291.
[12] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 35–49.
[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.
[14] Jiang Jinghui, Huang Zhenpei, Xiang Qiao, Tang Lu, and Shu Jiwu. 2022. *P4-DPLL: Accelerating SAT Solving Using Switching ASICs*. Technical Report. Xiamen University.
[15] Kenji Kanazawa and Tsutomu Maruyama. 2010. An approach for solving large SAT problems on FPGA. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4, 1 (2010), 1–21.
[16] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*. 1–12.
[17] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with {In-Network} Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.
[18] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. 2016. Be Fast, Cheap and in Control with {SwitchKV}. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 31–44.
[19] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.
[20] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 795–809.
[21] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 143–157.
[22] João P Marques Silva and Karem A Sakallah. 2003. GRASP—a new search algorithm for satisfiability. In *The Best of ICCAD*. Springer, 73–89.
[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[24] Anh Hoang Ngoc Nguyen, Masashi Aono, and Yuko Hara-Azumi. 2020. FPGA-based hardware/software co-design of a bio-inspired SAT solver. *IEEE Access* 8 (2020), 49053–49065.
[25] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).
[26] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 179–193.
[27] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 126–138.
[28] Yang Zhang, Bo Han, Zhi-Li Zhang, and Vijay Gopalakrishnan. 2017. Network-assisted raft consensus algorithm. In *Proceedings of the SIGCOMM Posters and Demos*. 94–96.
[29] Peixin Zhong, Margaret Martonosi, and Pranav Ashar. 2000. FPGA-based SAT solver architecture with near-zero synthesis and layout overhead. *IEE Proceedings-Computers and Digital Techniques* 147, 3 (2000), 135–141.
[30] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *arXiv preprint arXiv:1904.08964* (2019).
[31] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1225–1240.