TECHNIQUES FOR HIGH PERFORMANCE MATCHING

A Dissertation

by

PING WANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Alex Sprintson
	Krishna Narayanan
	Riccardo Bettati
Head of Department,	Miroslav M. Begovic

August 2021

Major Subject: Electrical and Computer Engineering

ABSTRACT

With the growth of big data application demands, improving high-performance computing (HPC) becomes an essential industry task. High-performance matching is a critical performance path for HPC communications because it significantly impacts computing performance and profoundly affects networking performance. This dissertation focuses on improving the high-performance matching in HPC networks to keep up with the increasingly heavy demands of evolving applications.

This dissertation is tackling the matching problem from both the computational and network aspects. On the one hand, the Message Passing Interface (MPI) is a *de facto* standard for the communication of parallel processes in an HPC network [1]. MPI has delivered an excellent performance for running large-scale scientific applications in petascale systems. Along with the petascale system, the exascale system is evolving to run even larger applications where the computing job size increases dramatically. This trend enlarges the message queues and degrades the MPI message matching performance. With the increasing requirement of big data applications, MPI message matching is a critical performance path for HPC communications. On the other hand, with the blooming of network techniques and the fast-growing size of network applications, users are seeking more enhanced, secure, and various network services. In an HPC network, the HPC cluster comprises multiple interconnected nodes in a switched network. With the integration of software-defined networking (SDN) technology into the HPC network, both the computational and network resources can be allocated efficiently according to the applications' requirements. Thus, SDN switches are deployed in HPC networks to support high-performance, differentiated network services and guarantee the diverse users' needs, such as firewall, load balancing, and quality of service [2]. In an SDN switch, packet classification classifies incoming packets to flows according to the rules generated in the control plane, which is a switch's core function. Therefore, packet classification becomes a critical performance path for the HPC network.

First, this dissertation presents GenMatcher, a generic and software-only arbitrary matching

framework for fast and efficient searches on packet classification. The goal is to represent arbitrary rules with efficient prefix-based tries. In order to generate efficient trie groupings and expansions to support all arbitrary rules, we propose a clustering-based grouping algorithm to group rules based upon their bit-level similarities. Our algorithm generates near-optimal trie groupings with low configuration times and provides significantly higher match throughput than prior techniques. Experiments with synthetic traffic show that our method can achieve a 58.9X speedup compared to the baseline on a single-core processor under a given memory constraint [3][1].

Second, to further improve the GenMatcher performance, this dissertation proposes GenSMatcher, an efficient Single Instruction Multiple Data (SIMD) and cache-friendly arbitrary matching framework. GenSMatcher adopts a trie node with a fixed high-fanout and a varying span for each node depending on the data distribution. The layout of the trie node leverage cache and modern processor features such as SIMD instructions. To support arbitrary matching, we interpret arbitrary rules into three fields: value, mask, and priority, and then propose the GenSMatcher extraction algorithm to process the wildcard bits to support randomly positioning wildcards in arbitrary rules. At last, we add an array of wildcard entries to the leaf entries, which stores the wildcard rules and guarantees matching results. Experiments show that GenSMatcher outperforms GenMatcher under a large scale of the ruleset and key set regarding search time, insert time, and memory cost. Specifically, with 5M rules, our method achieves a 2.7X speedup on search time, and the insertion time takes ~ 7.3 seconds, gaining a 1.38X speedup; meanwhile, the memory cost reduction is up to 6.17X.

Third, to guarantee MPI ordering feature and high-performance matching for big applications on MPI tag matching, this dissertation introduces a new hybrid data structure and matching mechanism to address the performance challenges, reducing the matching operation time in the posted receive queue (PRQ) and unexpected message queue (UMQ). The hybrid data structures are composed of tries and hash maps. We evaluate our mechanism on microbenchmarks

[1]This paragraph is reprinted with permission from GenMatcher: A Generic Clustering-Bashed Arbitrary Matching Framework by Ping Wang, Luke McHale, Paul Gratz, Alex Sprintson, 2018. ACM Transactions on Architecture and Code Optimization, Volume 15, Issue 4, Article No. 51, https://dl.acm.org/doi/10.1145/3281663.

and existing MPI applications with different numbers of processes. Experiments with synthetic message flow show that our method can achieve a 20X search time speedup compared to the single-core processor's baseline. For the PICSARlite application, we integrated our Hybrid and Intel mechanism into the MPICH library and evaluated their performance on the Ada cluster of Texas A&M University, which has 793 general compute nodes. The experiment outcome shows that our proposed Hybrid mechanism can achieve up to 1.55X speedup compared to the MPICH library method.

DEDICATION

To my husband, Derek, and my loving parents

ACKNOWLEDGMENTS

First and foremost, I would like to sincerely thank my advisor, Paul V. Gratz, for his endless support, advice, and insight. Paul always kept faith in me, guided me to explore the research problems from various aspects, and mentored and encouraged me to tackle them. I would not complete my research without Paul's advice and support.

I would also like to thank my committee members, Alex Sprintson, Krishna Narayanan, and Riccardo Bettati, for their helpful feedback on my research. During my time at TAMU, I was very fortunate to have been a part of the Internet 2 Technology Evaluation Center. I want to thank Walt Magnussen for the two-years funding support and public safety experience. Also, I would like to thank the Texas A&M High Performance Research Computing Center. Thanks for the environment and techniques support.

I would like to acknowledge my colleagues at Computer Architecture, Memory Systems and Interconnection Networks (CAMSIN). I was very fortunate to work with you all. Special thanks to Mian Qin and Luke McHale. Thank both of you for always helping me debugging and discussing issues in my research.

Finally, I would like to thank my husband and my parents, who have unconditionally supported me during my Ph.D. journey. I am very fortunate to have all your love and support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Paul V. Gratz, Professor Alex Sprintson, and Professor Krishna Narayanan of the Department of Electrical and Computer Engineering and Professor Riccardo Bettati of the Department of Computer Science and Engineering.

Chapter 5 was collaborate with Pavel Shamis of ARM Inc.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by the teaching assistant fellowship from Texas A&M University.

NOMENCLATURE

PRQ	Posted Receive Queue
UMQ	Unexpected Message Queue
HPC	High Performance Computing
SDN	Software Defined Network
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions
MPI	Message Passing Interface
OGAPS	Office of Graduate and Professional Studies at Texas A&M University
B/CS	Bryan and College Station
TAMU	Texas A&M University
TCAM	Ternary Content-addressable Memory

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION*

1.1 High-Performance Matching

Over the past decades, with the increasing demands of high-performance data centers, there has been a steady evolution in the high-performance computing (HPC) industry. To process the large volume applications and achieve higher performance, the interconnect is the enabling technology. Traditionally, a high-performance compute cluster consists of a single-node CPU, which has been evolved to multi-nodes across the entire cluster. Furthermore, the processor has been developed from single-core to many-core. To keep pace with the heavy demands for Terascale performance, Petascale performance, and even Exascale performance in the industry, the companies must keep seeking new techniques to improve performance and satisfy applications' needs.

High-performance matching is a critical performance path for HPC communications because it has a significant impact on computing performance and has profound effects on networking performance. This dissertation focuses on improving the high-performance matching in HPC networks to keep with the heavy demands for growing applications. The data center is composed of three parts: compute, networks, and storage. The dissertation is tackling the matching problem in both compute and network aspects. From the computing aspect, the computing job size grows with the growth of Exascale computing. This trend causes the message passing interface (MPI) message queue to grow, degrading the MPI tag matching performance. Thus, MPI message matching is a critical performance path for HPC communications. From the networking aspect, users have required high demand for more secure, reliable, and various network services as the network technology keeps growing fast and network applications retain emerging. Packet classification is a critical component in a switch network. Moreover, with SDN and cloud computing

*Section 1.1.1 is reprinted with permission from GenMatcher: A Generic Clustering-Bashed Arbitrary Matching Framework by Ping Wang, Luke McHale, Paul Gratz, Alex Sprintson, 2018. ACM Transactions on Architecture and Code Optimization, Volume 15, Issue 4, Article No. 51, https://dl.acm.org/doi/10.1145/3281663.

proposed, the Internet requires high-performance packet classification of multi-fields. Therefore, the packet classification in a switch is a critical component for HPC communications.

1.1.1 Packet Classification

Packet classification is an enabling function for a variety of applications within networking, including Quality of Service (QoS), security, monitoring, and multimedia communications. The emerging distributed computing and big data applications require impose strict requirements for data processing delays and throughput. On the other hand the emergence of Software Defined Networking (SDN) and a push towards more general purpose networking hardware is driving the need for optimized software approaches to high throughput matching. Accordingly, in this dissertation we present a generic and efficient pure-software mechanism for arbitrary matching. Our algorithm can be used in a broad range of packet classification application, including SDN packet processing pipeline.

In general networking applications, packet classification typically includes bit-wise matching of a *key* against a pre-defined rule set [4, 5, 6, 7, 8]. The key is typically a a subset of packet header fields, but might also include other meta-data derived from the packet header.

The matching functions used in packet classification typically come in four forms:

- **Exact match:** An exact match rule corresponds to exactly one matching key (*i.e.* no wild-card bits).
- **Range match:** A range match rule defines a sequential set of possible matching keys. This form of matching is occasionally used in defining a rule which covers multiple matching ports (*e.g.* from $1 - 30$ inclusive).
- **Arbitrary match:** An arbitrary match rule contains wildcards (*i.e.* a wildcard bit can be either 0 or 1) at any bit position, matching 2^n possible keys for each wildcarded bit.
- **Prefix match:** In prefix matches, all the wildcards must uniformly cover the low-order bits of the rule.

While there exists a large body of research on packet matching, most of this work focuses on prefix and range matching. With the developing trends towards new SDN services, big data, and High Performance Computing (HPC) matching is no longer performed with five standard header fields. New classification applications examine additional fields, increasing rule flexibility. Further, parallel computing techniques for machine learning and big data analytics are blooming and can process increasing volumes of data simultaneously. Thus, more generic packet matching methods are required to deal with all the various matching requirements[1].

Prefix matching has been most heavily studied in prior work [9]. An essential improvement for software-based prefix matching is the trie data structure [9]. A trie is a binary tree data structure that significantly accelerates matching by bounding the search complexity to the tree depth instead of the total rule count.

While tries can significantly accelerate matching for situations where prefix matches are defined, ultimately, this approach is insufficient to meet all matching needs. In particular, tries cannot be directly used in non-prefix, arbitrary match scenarios where wildcards may appear at any bit-position. Unfortunately, as we will describe, in most current applications, while prefix matches may be defined for particular fields of a rule when multiple fields of type other than exact match are defined in a rule, the problem becomes one of arbitrary matching. Accordingly, this dissertation proposes a software framework for arbitrary matching, which considers the performance trade-off between search time and memory cost. This framework aims to achieve the highest search throughput, under a given memory constraint, for generic arbitrary matching.

We propose $GenMatcher$, a generic clustering-based arbitrary matching framework and a software-based arbitrary matching approach. The basic idea of $GenMatcher$ is to transform the ruleset into a minimal (under a given memory bound), optimized set of prefix format rule groups. All rules are carefully organized into a minimal number of groups, such that each group constructs a trie – transforming the group's subset of arbitrary match rules into prefix match rules.

[1]Range matching can be trivially transformed into a small number of prefix matches. Thus we do not discuss it further.

Creating a minimal set of prefix rule groups from the initial arbitrary rules requires a careful transformation. $GenMatcher$ [3], uses a correlation clustering-based grouping algorithm. To enable this clustering grouping algorithm, GenMatcher finds the relationships among all rules and allocates them into separate groups according to their similarities. For a given number of rules N, the GenMatcher grouping algorithm's complexity is $O(N^2)$. This performance is an enormous improvement versus the complexity of a brute force search for the optimal grouping, which we show has a complexity of $O(N^N)$.

After GenMatcher finds the best grouping of rules, we employ a bit swapping algorithm based on prior work [10] to rearrange the bit order, such that the largest possible prefix match is constructed. After the bit swapping operation, if there are any rules with wildcard bits outside of the prefix, we expand these rules to ensure all rules become prefix rules. Because of this rule expansion, memory may rapidly exceed the available memory in the system. Thus, one objective of $GenMatcher$ is to improve search time under a given memory bound. Finally, $GenMatcher$ inserts these transformed prefix rules into trie data structures, one trie per group. Then for each incoming key, we traverse all the tries to find a match.

Challenges: $GenMatcher$ employs its grouping algorithm to allocate all the rules into minimal groups under a given memory constraint threshold. In general, clustering [11, 12, 13, 14, 15, 16, 17, 18] is a powerful tool for finding the underlying structure of a large data set [19, 20]. Our goal is to apply the clustering result to build a data structure for efficient arbitrary matching. To implement the efficient arbitrary matching, we must address several challenging technical problems:

- We need to determine a similarity function to correlate and group the rules.
- We need to develop a grouping algorithm to maximumly group similar rules together based on the similarity function.
- We must guarantee that the grouping algorithm will produce a data structure that can satisfy performance requirements while remaining within a given memory constraint.

Contributions: $GenMatcher$ is a generic arbitrary matching approach that can deal with arbitrary rules without requiring specified human hints or other configuration metadata. Our $GenMatcher$ has the following features:

- We introduce $GenMatcher$, a generic arbitrary matching framework, which can process any form of matching used in packet classification. GenMatcher is the first framework we know for optimizing general, bit-wise arbitrary matching under a given memory bound.
- We develop a novel similarity function for use in correlation clustering-based grouping. This similarity function is the first function to be used in arbitrary matching to the best of our knowledge. The similarity function is a bit-wise based, efficient means to extract the relationship between rules. Any two rules in a rule table have a different similarity value based on the overlapping wildcard distributions. The similarity value is used to allocate all the rules into minimal groups under a memory constraint to balance search time with memory cost.
- Unlike prior arbitrary matching techniques, GenMatcher does not require specialized hardware. It is a pure software approach that can be applied to any general matching problem.

Since GenMatcher utilizes a binary trie data structure to improve performance further, we explore the optimization of trie data structure. Thus, we propose GenSMatcher, an efficient SIMD and cache-friendly arbitrary matching mechanism. GenSMatcher can take advantage of the modern processor features.

1.1.2 MPI Tag Matching

MPI is a *de facto* standard for the communication of parallel processes in High-Performance Computing (HPC) network [1]. It defines how data is moved from the address space of one process to another. HPC network requires the provider to move data faster among the various HPC cluster nodes and eliminate wasted compute time. The MPI standard defines how the processes communicate with each other via tag matching operations. Because of this, the tag matching performance plays a crucial role in HPC performance.

The emerging HPC applications require the imposition of strict requirements for data processing delays and throughput. Accordingly, in this dissertation, we present a hybrid data structure for MPI tag matching to improve MPI communication. The message envelope is used to distinguish messages whose information consists of a tuple of fields: $\{contextID, rankID, tag\}$, represented by (c, s, t). A receive operation can receive a sending message if its envelope matches the source, tag, communicator values specified by the receive operation [21].

As shown in Figure 1.1, there are two tables used for matching. One table is posted receive queue (PRQ), which stores the message envelop specified by the receive operations. The other one is the unexpected message queue (UMQ), which stores the non-matched message envelop from the sending operations. Send operation and receive operation are triggered by MPI_Send( ) and MPI_Recv( ), respectively. The two operations are asynchronous and independent. In Figure 1.1, the solid line represents the procedure for the send operation. The dashed line represents the receive operation's procedures.

The sending message searches through the PRQ and checks if there is a matching received message. If it is a match, the matched receive message will be deleted from the PRQ, and the sending message payload will be stored into the buffer specified by the received message. If it is a non-match, the sending message will be inserted into the UMQ. For the receive operation side, when a new receive operation arrives, it will search through the UMQ to check if there is a matching sending message. If it is a match, the matched entry will be deleted from UMQ, and the corresponding send message's payload will be stored into the buffer specified by the received message. If it is a non-match, the received message will be inserted into the PRQ.

Note, there are three operations for both PRQ and UMQ: search, delete, and insert. The two tables' fields are represented by (p, c, s, t), which stands for message sequence ID (priority), communicator index, source rank ID, and tag value. MPI messages are matched using three fields (c, s, t). P is used for keeping the order of messages, which guarantees the order semantic. At the end of application processing, all the send messages and receive messages are matched perfectly.

In Figure 1.1, we can see that all the values of each field for send messages are specified

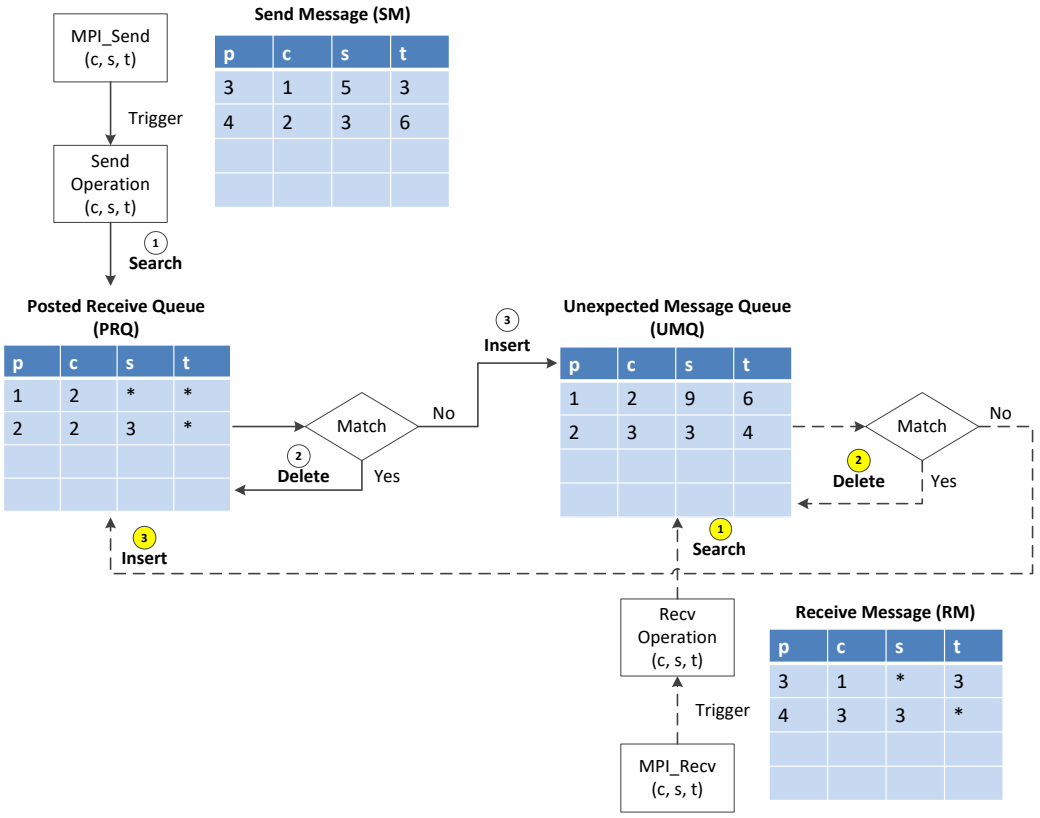


Figure 1.1: Tag matching framework

numbers. However, for receive messages, there may have a wildcard on the field source or tag. This is the reason why we need to guarantee the order semantic. Since the received message can have a wildcard on-field source and tag, there might have multiple matches between send messages and receive messages. The semantic order rules guarantee that the pair with the highest priority sending and receiving messages will always match the result when there are multiple match candidates. Thus, to guarantee correct application processing, the match result always needs to be the highest priority.

The challenge of MPI tag matching is to achieve high performance while guaranteeing the order semantic. Currently, the linked list is a traditional data structure to store all the messages and guarantee the communications between processes across cores run successfully. However, as the applications scale up, the linked list's length becomes very large, and the search performance is significantly degraded. Thus, improving the tag matching performance is a crucial problem.

This dissertation proposes a hybrid data structure combined with a trie and hash map, which can process the wildcard messages efficiently and improve the matching performance.

1.2 Dissertation Statement

In this dissertation, we introduce three application-driven arbitrary matching mechanisms that can improve the matching performance. First, We propose GenMatcher, a generic, software-based arbitrary matching framework. All rules are represented in exact or prefix format and inserted into a binary trie. Second, we explore the SIMD and cache-friendly data structure and develop a SIMD-based arbitrary matching mechanism, namely, GenSMatcher. GenSMatcher outperforms GenMatcher and achieves up to 2.7X search time speedup. Finally, we propose our Hybrid MPI tag matching mechanism in order to improve the HPC matching performance.

1.3 Dissertation Organization

In the remaining chapters, Chapter 2 summarizes the current matching mechanism in terms of software and hardware aspects. Chapter 3 introduces a generic clustering-based arbitrary matching framework (GenMatcher). Chapter 4 extends the GenMatcher and proposes a generic SIMD-based arbitrary matching mechanism (GenSMatcher). Chapter 5 introduces a hybrid message matching mechanism for HPC communications and shows how this new mechanism can effectively improve the matching performance. Finally, Chapter 6 concludes this dissertation.

2. BACKGROUND*

This chapter presents a background of matching mechanisms. We summarize the current technologies for generic matching from both software and hardware aspects. From the software side, we mainly focus on data structures. From the hardware side, we consider cache efficiency and data-level parallelism.

In this dissertation, we study the matching operations at bit-level. The matching operations are composed of two objects. One is the ruleset that builds the database. The other is the key set, which traverses the database and tries to find a match. For different applications, the properties/distributions of the ruleset and key set might be different. However, from the bit-level view, all the data are bit-0, bit-1, or bit-$*$[1]. The matching operations in various applications typically come in four forms: Exact match, Range match, Arbitrary match, and Prefix match, where the arbitrary match is the generic form that covers all the different matching types. After determining the matching types, we present the existing matching mechanisms from both the software and hardware aspects.

2.1 Data structures

We build the database by inserting all the rule sets into a data structure to search into and implement the matching operations. This section introduces all the common data structures and explains how to implement the matching operations using these data structures.

2.1.1 Array

An array is the most straightforward data structure to store the data set. For the insertion operations, all the rules are inserted into the array. The time complexity is $O(N)$, where N is the

*Section 2.2.2 is reprinted with permission from GenMatcher: A Generic Clustering-Bashed Arbitrary Matching Framework by Ping Wang, Luke McHale, Paul Gratz, Alex Sprintson, 2018. ACM Transactions on Architecture and Code Optimization, Volume 15, Issue 4, Article No. 51, https://dl.acm.org/doi/10.1145/3281663.

[1]The symbol $*$ denotes the wildcard bit, which can be either bit-0 or bit-1.

number of rules. For the search operations, the time complexity is $O(N)$. If N is a large number, the matching performance will be getting worse. To improve the search performance, next, we present the trie data structure.

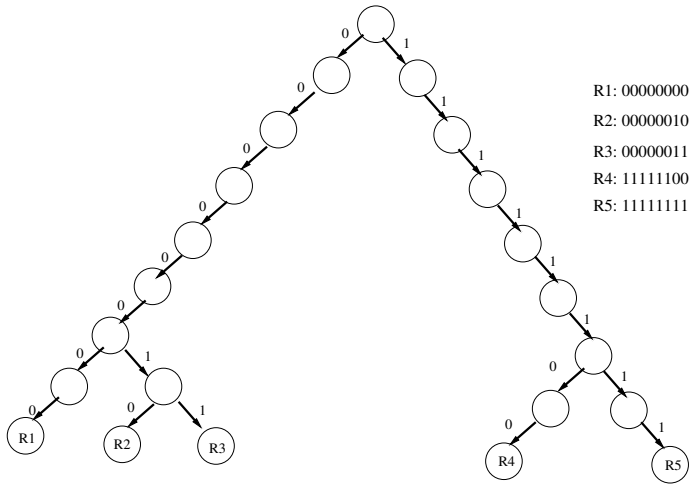


Figure 2.1: Binary trie data structure.

2.1.2 Binary trie

Tries are tree data structures that a node's children share a common prefix. That is, the trie data structure stores the rules by its digital representation [22]. If a rule is an integer, each bit of the rule is inserted into the trie. If a rule is a string, each char character will be inserted into the trie. The trie is an order-preserving structure, which is the unique feature of its data structure. A $binarytrie$ is the basic trie structure, in which each node can have at most two children, known as the left child and right child. Compared to linear search via an array, the trie search's time complexity is $O(h)$, where h is the trie height, typically determined by the length of the rule's digital representation. In general, trie has a better performance than $O(N)$. Suppose a rule is an 8-bit integer whose bits are inserted into the trie following the bit-order during the insert process. Bit-0 will be inserted into the left child path, and bit-1 will be inserted into the right child path. Thus, the trie height will be 8. During the search process, for each 8-bit key, it searches through

the entire trie to find if there is a match. Thus, the search might be slow due to a considerable tree height.

Figure 2.1 shows an example of a binary trie. In this binary trie data structure, five rules are inserted into the trie. $R1$, $R2$, $R3$, $R4$, and $R5$ are leaf nodes. All other nodes are internal nodes. Each node has at most two children. For an 8-bit rule, each bit is stored into a trie node. The figure shows that the height of the trie is 8. Accessing a trie node generally produces at least one cache line fill [23]. Therefore, the lower bound of the counts of cache line fills is the height of the trie. In the worst case, the searching process needs to traverse from the root node to the leaf node. To improve the search performance, reducing the trie heights and the number of cache line fills is needed. Next, we will introduce the Binary Pactricia [24] trie.

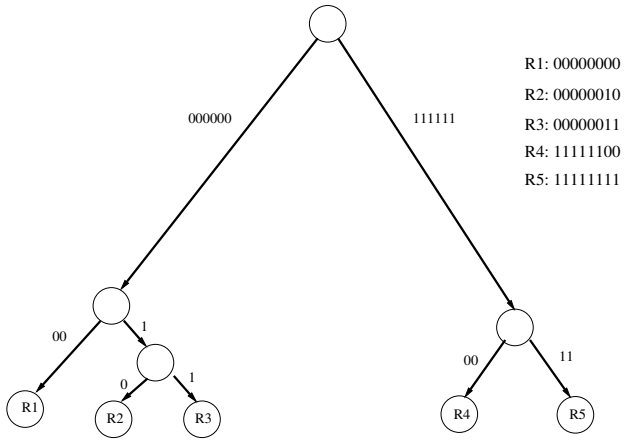


Figure 2.2: Binary Pactricia trie data structure.

2.1.3 Binary Pactricia trie

In Figure 2.2, a binary Patricia trie is depicted. From the figure, we see five rules stored in this binary Patricia trie, which has four inner nodes (including the root). Thus, binary Patricia trie has the property: $N - 1$ inner nodes and N leaf nodes, where N is the number of rules stored in the trie data structure. In contrast to the binary trie, the binary Patricia trie cut down the trie

height by bypassing the nodes with a single child (left child or right child) [25]. The key idea is to compress the rule path. For a long path where the nodes have only one side child, we can compress the multiple nodes into one single node. Compared with Figure 2.1, the trie height is reduced from 8 to 3. However, since this Patricia trie is still in binary, the Patricia trie still has a significant tree height with a more extensive data set. Therefore, we introduce the M-ary trie next.

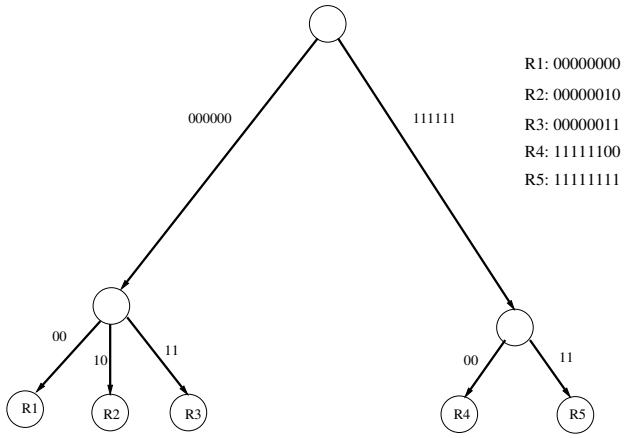


Figure 2.3: M-ary Pactricia trie data structure.

2.1.4 M-ary trie

In order to further decrease the trie height, we enlarge the span from 1-bit to M-bit. Here we take $M = 2$, where M is the span, and 2^M is the upper bound of the counts of children of each trie node, known as the fanout. As shown in Figure 2.3, the children of each node have bit-00, bit-01, bit-10 and bit-11. In comparison with Figure 2.2, we see that the overall trie height is reduced from 3 to 2.

Although increasing the span can reduce the trie height, there can still be downsides. The increased span decreases the trie height linearly while the memory cost increases exponentially [26]. Since the span is fixed, each node is allocated with 2^M pointers in an array, no matter what the data distributions are. Thus, for sparsely distributed rules, a large amount of memory

is wasted given that the majority of the pointers allocated to the nodes are empty. The number of children of each node, namely the fanout, is generally smaller than the value 2^M, especially in the lower level of a tree.

From the SIMD instruction aspect, modern CPUs allow performing multiple comparisons using a single SIMD instruction. M-ary trie can make use of SIMD instructions to reduce the the number of comparison. From the aspect of cache efficiency, since increasing M decreases the number of cache line fills, then the number of cache misses is reduced. This improves cache efficiency. However, it also enlarges the node size, leading to more wasted memory. Thus, it is vital to find the sweet spot between the search performance and memory cost. Next, We introduce some trade-off data structures.

2.1.5 Adaptive radix trie

Considering the above traditional binary search trie data structures or M-ary trie, they cannot make full use of the modern hardware features since they do not allow for cache utilization and SIMD utilization optimally. [26] proposes an adaptive M-ary radix tree, whose maximal fanout can go up to 256 children. This paper utilizes four different node types to reduce memory cost and improve cache efficiency depending on data distribution. Also, it utilizes SIMD instructions to improve the search performance via data-level parallelism.

Figure 2.4 represents an adaptive radix trie (ART), where each rectangle is a trie node that has a different node size according to their children counts. If a node has many children, we assign the node with a bigger size. Otherwise, the node is allocated to a smaller size. You can see that a larger rectangle has more pointers that point to its respective children in the figure. The ART node is configured with a relatively large span and a varying fanout depending on the corresponding node size, balancing the search time performance and memory cost. Since the trie's height is determined by the rule length and the span size, the height is not affected by the adaptive node's design. However, the adaptive nodes lead to a more negligible memory cost by reducing the nodes' sizes. Consequently, given a constrained memory size, we can choose a larger span, resulting in a smaller height. Hence, ART enhances the search performance along with the

memory cost. Because of the fixed span size, the trie height is still determined by the rule length. Thus, the ART is an unbalanced trie when the data is sparsely distributed.

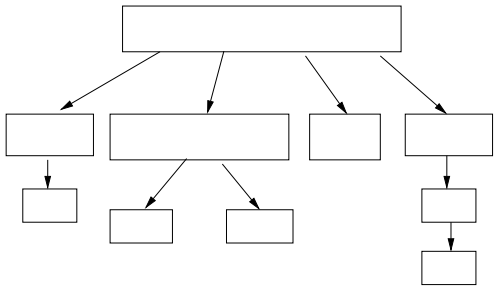


Figure 2.4: ART data structure.

For all the tries mentioned earlier, the trie node's span size is a fixed value. However, nowadays the big data are all application-driven. Various applications have different data distributions. Some big data is more sparsely distributed. That data is inserted into a trie, in which each level has a different data size. If a fixed span size is used, there will be a large amount of memory wasted. In order to alleviate this impact from sparse data, we must build a flexible data structure suitable for data-dependent applications.

2.1.6 Height optimized trie

Height Optimized Trie (HOT) [25] is a new data structure designing for a main-memory database with high performance and low memory cost. Unlike each node in ART with a fixed span size and various fanouts, HOT proposes that each node has its respective span size depending on the data distribution. For a conventional trie, it has an exact span and data-dependent fanout. However, HOT produces a data-dependent span[2] and an explicit maximum fanout[3] k. In the details, HOT proposes a composite node composed of k binary Patricia trie node. Thus, each composite node has a fixed maximal number of children/leaf Binary Node, but each child may

[2] $Span$ is defined as the number of bits of a node, e.g., the binary trie has a span of 1.

[3] In any tree data structure, the fanout of a node is defined to be the number of children the node has. The fanout of a tree is defined to be the maximum fanout of any node in the tree.

involve different bit positions. That is, each compound node has a different span size and span bits.

To trade off the memory cost and search performance, HOT stores the partial rules extracted from the discriminative bits, which are the discriminating bits between the different rules. Therefore, the memory cost is decreased, especially for the sparse data. We demonstrate a ruleset in Figure 2.5. Each rule is an 8-bit data and HOT insert rules based on the discriminative bits between different rules stored in the trie. We index the bit from the least significant bit to the most significant bit. Thus, the rightmost bit is bit-0, and the leftmost bit is bit-7.

R1: 00000000
R2: 00000010
R3: 00000011
R4: 11111100
R5: 11111111
R6: 11011111
R7: 00010000

Figure 2.5: An example of a rule set.

For building the trie, HOT insert rules one by one. Before any insertion, HOT traverses the trie until the compound node with the missing match *BiNode* [4] is found. Here we assume $k = 3$ such that each compound node can have up to 3 leaf children nodes. For the first rule $R1$, they insert the rule into the root since the trie is empty and there is no comparison. For the second rule, $R2$, because the trie is not empty, before inserting $R2$, HOT needs to traverse the current trie data structure to find the missing match *BiNode* in a compound node. For this case, we compare $R2$ 00000010 with $R1$ 00000000 and find the missing match *BiNode* bit-1. Therefore, a first new discriminating *BiNode* bit-1 is constructed and added into the affected compound node as long as the compound node has less than k leaf entries after insertion. For the third rule $R3$, HOT

[4] *BiNode* is a binary Patricia node in a compound node.

traverse the trie first and find the missing match bit-0. Next, a second new discriminating *BiNode* bit-0 is generated and placed in the dedicated compound node.

For rule $R4$, the new discriminative bit is bit-7, as shown in Figure 2.6 (c), this compound node already has three leaf entries. Thus, a new compound node is created, as shown in Figure 2.6 (d). The Rule $R5$ generates the discriminating bit-1 under the bit-7 path, as shown in Figure 2.6 (e). For rule $R6$, it generates the new discriminative bit-5 and a new compound node. Finally, for the last rule $R7$, it creates discriminative bit-4 shown in Figure 2.6 (g).

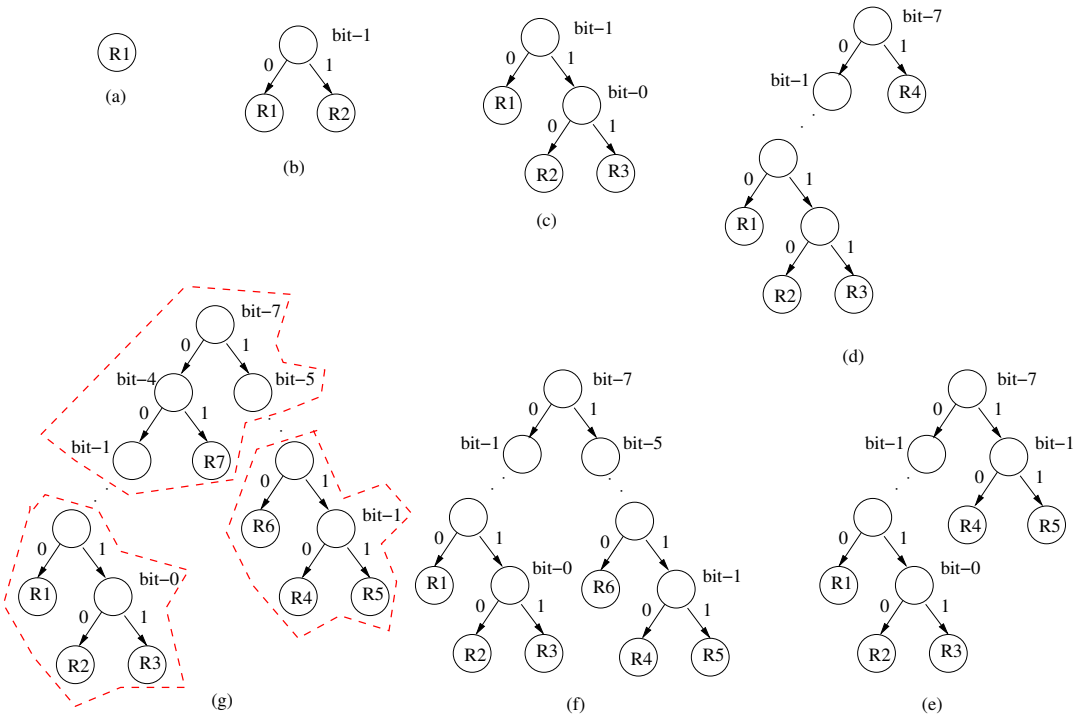


Figure 2.6: An example of construction of HOT data structure.

In Figure 2.6 you can see the insertion process of these seven rules. From the Figure 2.6 (g) note that each rule has its respective span bits. $R1$ involves bit-7 bit-4, and bit-1. $R2$ and $R3$ have bit-7, bit-4, bit-1, and bit-0. $R4$ and $R5$ have bit-7, bit-5, and bit-1. $R6$ involves bit-7 and bit-5. $R7$ involves bit-7 and bit-4. Thus, for this rule data set, the discriminative bits set is { bit-7, bit-5, bit-4, bit-1, bit-0 }. Only its associated discriminative bits are inserted in the trie data structure

to save each rule's memory cost. As shown in Figure 2.6, the *BiNode* is represented by the circle, and the red area is a compound node with the maximal three-leaf entries.

In summary, the insert process is as follows:

1. Extract the discriminative bits from the new rule that needs to be inserted, namely, partialRule.
2. Utilize SIMD instructions to compare the partialRule with each compound node's inserted entries level by level.
3. Find the matched entry index from the leaf node (compound node).
4. Utilize executeForDiffingKeys( ) to check if the matched entry is the same as the inserting Rule. If it is true, get the new discriminative bit and get the insertion information. Otherwise, the inserting rule will be abandoned since the value is already in the trie.
5. Follow the k-constraint rule to insert the new BiNode and the value in the leaf node.

For the search process, HOT proposes the following steps:

1. The partial key is extracted from the incoming key via the Discriminative bits.
2. Utilize SIMD instructions to compare the partial key with each compound node's inserted entries level by level.
3. Find the search result index from the leaf node (compound node).
4. Since HOT only inserts partial keys into the trie data structure, after finding the matched index of the leaf node, the whole rule value in the leaf node needs to be compared with the full incoming key to guaranteeing the matching is correct.

We conclude the properties of HOT data structure as follows:

- The height/complexity of HOT depends on the length of the rules and the relations among all the rules since they build the trie using the discriminative bits between the new rule and the current trie data structure.

- HOT only stores partial rules utilizing their discriminative bits in their *BiNode*.
- The partial rules are stored in lexicographic order.
- The path to a leaf node represents the partial rule of the whole rule stored in the leaf node. Thus, rules need to be stored in the leaf node and guarantee the correct match result.
- HOT does not require re-balancing operations, and any insertion orders of rules result in the same trie.
- HOT utilizes SIMD instructions to process the data in parallel.

The goal of HOT is to optimize the trie height in order that a maximum number of partitions along a path from the root node to any leaf node is minimized [25]. Because of the properties of HOT, it is an ideal data structure to perform arbitrary matching due to its smaller trie height and memory cost. Also, its node layout gets the benefits from cache efficiency and SIMD-optimized search.

2.1.7 Hash table

Hash tables are another famous data structure for matching operations since hash tables have the search time complexity $O(1)$, which is much faster than the tree search time complexity $O(\log_2 N)$, where N is the ruleset's size. The search time complexity is faster than the trie search time complexity $O(h)$, where h is the trie's height. Although the search time is much faster, it only supports the exact match. It cannot support arbitrary matches. Also, we need to consider the collision problem when the data set is large. Therefore, arbitrary matching cannot use the traditional hash tables to improve the matching performance.

Trie data structure is ideal for arbitrary matching applications. This is because the application is generally significant. Although the trie data structure's search time complexity is not as fast as $O(1)$, with the increasing size of the data set, the time complexity of trie $O(h)$ is independent of the data size.

2.2 Hardware

For the hardware side, we consider the SIMD utilization and Ternary Content-Addressable Memories (TCAMs) implementation. Modern CPUs support simultaneous comparisons in parallel using a single SIMD instruction [26].

2.2.1 SIMD instruction set

Modern Processors are featured with wide vector units, known as SIMD [27], exploiting data-level parallelism. Intel's Streaming SIMD Extensions expand the register width from 32-bit to 128-bit, to support processing four single-precision floating-point numbers at one time. The subsequent Intel Advanced Vector Extensions (AVX) and Intel AVX2 was further extended to process 256-bits of data at one time. The successor Intel Advanced Vector Extension 512 is able to operate on 512-bits of data with a single instruction.

SIMD instructions are supporting various operations. This dissertation mainly uses comparing operations, bit operations, and register I/O operations. In contrast to comparing scalar types' operations, there cannot be a single true or false result for a SIMD comparison. Instead, there will be multiple boolean values in the result. The register width determines the size of this vector of boolean values. A vector of boolean values is named a *mask*. For bit operations, we utilize boolean logic operations, selective bit moving operations, bit counting, and bit extract operations. Concerning the register I/O operations, we utilize broadcast load operations.

In this dissertation, we utilize SIMD instruction to improve the performance in search performance and memory cost.

2.2.2 TCAMs

The TCAMs [28] represent a hardware-based approach to matching. A TCAM is a specialized memory array with integrated comparison logic, where each entry stores a rule that is encoded in a ternary format (*i.e.* 0, 1, and $*$). Thus, arbitrary rules can be stored in a TCAM directly. The TCAM can compare keys against all stored rules in parallel and return the match result with the highest priority. TCAMs take $O(1)$ time to finish the search and generate a match result.

However, TCAM cells require much higher area and power than traditional static random access memory (SRAM) arrays. As a result, TCAMs are expensive and often have limited capacity. Nevertheless, TCAMs have become the industrial standard for high-throughput packet classification [29], which takes a significant role in matching applications. However, TCAM parameters, such as rule length and number of entries are determined at the hardware design time, often making them a poor fit for the SDN paradigm of network application definition/configuration at runtime.

2.3 Conclusions

This chapter summarizes the current data structure and hardware design for matching operations. According to arbitrary matching features, we choose the trie data structure to implement the matching operation. In the following chapters, we will introduce our proposed trie data structures to improve the matching performance.

3. GENMATCHER: A GENERIC CLUSTERING-BASED ARBITRARY MATCHING FRAMEWORK*

3.1 Introduction

Packet classification methods rely upon packet content/header matching against rules. Thus, throughput of matching operations is critical in many networking applications. Further, with the advent of Software Defined Networking (SDN), efficient implementation of software approaches to matching are critical for the overall system performance.

Much prior work exists in both software and hardware based approaches to improve lookup performance for tables containing network IP addresses [30, 31]. IP lookup is typically defined as a composite prefix matching function, as illustrated in the left table in Figure 3.1. In this table there are five rules ($R1$-$R5$) which define the circumstances under which packets must undergo one of three actions (A, B, or C). Here, rules are defined using two fields ($Field1$ and $Field2$) where each bit is exact (0s or 1s in the table) or wildcarded (* in the table). If a given key is covered by rule's $Field1$ and $Field2$, then the key is considered matching and returns the defined action. Here, as is typical in networking applications, each field is a prefix match, thus all wildcards are constrained to the least-significant bit positions of each field.

Rules	Field 1	Field 2	Priority	Action
R1	10**	11**	1	A
R2	1***	101*	2	B
R3	00**	10**	3	A
R4	10**	1***	4	C
R5	100*	110*	5	B

Rules	Field 1	Priority	Action
R1	10**11**	1	A
R2	1***101*	2	B
R3	00**10**	3	A
R4	10**1***	4	C
R5	100*110*	5	B

Figure 3.1: Arbitrary packet matching.

*Reprinted with permission from GenMatcher: A Generic Clustering-Bashed Arbitrary Matching Framework by Ping Wang, Luke McHale, Paul Gratz, Alex Sprintson, 2018. ACM Transactions on Architecture and Code Optimization, Volume 15, Issue 4, Article No. 51, https://dl.acm.org/doi/10.1145/3281663.

3.1.1 Motivation

Arbitrary matching is the most general form of matching, covering all the other types of matching patterns, including the exact, range, and prefix match. Further, while traditional IP packet classification is typically defined in terms of prefix matches on individual field of the packet header, when multiple sets of such prefix defined match fields are defined together in each rule, the matching function becomes arbitrary. In Figure 3.1, each rule's *Fields* have different prefix masks defined (*i.e.* there are different numbers of *'s in each rule for each *Field*). While each field could be processed individually as a prefix match with a trie as described above, this would require a separate trie traversal for each field, with the match not determined until the results from all trie traversals are completed and combined. This approach restricts opportunities to optimize the data structures across the entire rule.

Alternately, matching might be accelerated by combining all *Fields* into a single match function, as shown on the right table of Figure 3.1. This approach means that the entire set of extracted header *Fields* can be compared against the rules in one operation, which does lend itself to optimizations as the entire rule is considered as whole. Doing so however, changes the matching function to arbitrary match as shown in Figure 3.1. Thus tries may not be used directly, motivating the need for new approaches to accelerate arbitrary matching.

We further note that arbitrary matches have applications beyond traditional network classification. With the emergence of big data, more and more applications need to leverage bit-wise matching. In HPC systems, the Message Passing Interface (MPI) standard defines a set of rules, known as tag matching [32, 33]. Tag matching is designed for matching source-send operations to destination-receives. Instead of matching the packet header of the switch network, there are many MPI message fields which must match a sender and its corresponding receive functions: 1). the communicator[1], 2). the user tag, including a wildcard specified by the receiver, 3). the source rank[2], including a wildcard specified by the receiver, 4). the destination rank. Thus, the

[1]In HPC system, the communicator is the process group ID in multi-core system.

[2]The rank is the stack address of a message that shows where the message comes from.

tag matching mechanism is a form of arbitrary matching.

3.1.2 Relationship with Prior Art

While prefix matching is heavily studied, arbitrary matching, particularly in software, remains a highly under-examined area. Among the few works in this area, Meiners *et. al* proposed Bitweaving [10], a non-prefix approach to compressing packet classifiers. We note that their approach was designed to compress rules for use with a ternary content-addressable memory (TCAM) [34]. [10] proposed a bit swapping algorithm to transform non-prefix rules into prefix rules. This was combined with a bit merging operation to reduce the number of rules. As this mechanism is intended to reduce the number of rules that need to be stored in a TCAM [35], the algorithm makes no further optimization to the rule set.

In this chapter, in order to avoid the hardware and energy cost of TCAMs, and to provide a general software-based solution, we propose a generic, software, arbitrary matching mechanism while keeping the door open to hardware acceleration.

3.2 Related Work

Prior work on arbitrary matching mainly falls into two categories: TCAM-based solutions and algorithmic solutions. The TCAMs [28] represent a hardware-based approach to arbitrary matching. TCAMs are generally unavailable in general purpose complexity. Significant research has been done on TCAM compression techniques [10, 36] to reduce the number of entries stored in TCAM. One such method defined by Meiners *et. al* is Bitweaving [10]. The primary aim of Meiners *et. al* in Bitweaving [10] was the compression and reformatting of rules such that they could be efficiently implemented in a TCAM. Their proposed technique is composed of a bit swapping algorithm and a rule merging algorithm. For bit swapping, first, they sort the rules by the number of wildcards in each rule. Second, they separate rules from the rule table into groups. Within each group, a single permutation is applied to each rule's bits to produce a reordered rule. After grouping, all the rules become prefix rules. They also propose a bit merging operation. In each group, if the hamming distance between any two adjacent rules is 1 then the rules will be

merged into one rule with bit $*$ on that position. For the hamming distance to be 1, there can only be one bit position difference between rules. After finishing the bit merge operation, the new merged rules will be processed by TCAM.

Bitweaving's intent was to produce rules which could be used in a TCAM, however, there is much to be learned from this work for the application of software-only arbitrary matching. In order to process any type of rules in our GenMatcher, we adapt Bitweaving's bit swapping algorithm to transform rules into a prefix format to the maximum extent. In the Bitweaving mechanism, all the rules are able to be transformed to a prefix format by applying a large swapping algorithm. We note that, critically, Bitweaving makes no attempt at expanding rules to minimize the number of groups. Thus, rules are broken into groups wherever a prefix match is not possible. Unlike Bitweaving, the objective of GenMatcher is to minimize the number of groups within a memory threshold, our technique expands the non-prefix wildcards to reduce the number of groups under a given memory threshold, which improves the search performance at some memory cost.

Although the approach here is intended for generic matching, there is much to be learned from prior work, software solutions based packet classification. He *et. al.* [37] proposed the *SmartSplit* algorithm. They split a large rule set in several subsets and using different packet classification algorithms for different subsets. Although this approach decreases memory consumption, as well as increases classification speed, it still suffers from rule duplication and cut decision configuration cost. Inoue *et. al.* [38] proposed multidimensional-cutting via selective bit-concatenation to accelerate many-field packet classification. However, the method they propose to generate lookup tables is complicated and there is no guarantee for a correct classification. Kogan *et. al.* [39] split the rules into two parts: independent order rules and order-dependent rules. The independent rules are divided into multiple groups, and the dependent rules are processed in the TCAM. Although they avoid impacting space and time complexities, it only focuses on range matching and requires a false positive test.

Other approaches [40, 41] perform packet classification on multi-core processors. Qi *et. al.* propose a technique which is able to support very large rule sets [40]. The approach, however,

suffers from memory expansion and can only support up to 1K 5-field rules. Qu *et. al.* [41] proposed an efficient bit-string aggregation technique to avoid excessive memory usage on multi-core processors. Lu and Sahini [42] propose a collection of hash tables to represent a multi-dimensional packet classification table. This work focuses on range matching. For a large size of rules and a many-field packet, the complexity of building the trie will become large. Several other software-based packet classification algorithms have been proposed which only deal with some specific rules or sub-rules but not the general form of the matching problem [37, 38, 39, 41, 42, 40]. By contrast GenMatcher is a generic arbitrary matching mechanism for any form of matching.

Several groups considered representations based on rule disjointness [5, 6, 43] and addressed efficient time-space tradeoffs for multi-field classification, where fields are represented by ranges. In these works, they assign all rules into multiple disjoint groups, where every group obeys a structural property on a subset of bit indices of a rule. Unlike their work, GenMatcher considers the rule as a whole, which results in a smaller time complexity for grouping.

3.3 GenMatcher

In this section, we present an overview of our design. GenMatcher is a generic clustering-based arbitrary matching framework for software packet processing, with the following design goals:

- **Performance**: It processes packets at high speed under a limited memory cost.
- **Generality**: It supports any type of bit-wise matching with rules of arbitrary length and wildcards.

To achieve these goals GenMatcher leverages prior work wherein efficient trie data structures are generated from sets of prefix rules. To this end, GenMatcher consists of 3 phases: map, group and build trie, as shown in Figure 3.2. The goal of these phases is to generate subsets of rules (groups), setup as prefix matches, from which tries can be generated through a combination of bit swapping (rearranging the bits in the rules) and rule expansion (wherein wildcards are enumerated). Our framework aims to keep the number of groups and amount of expansion

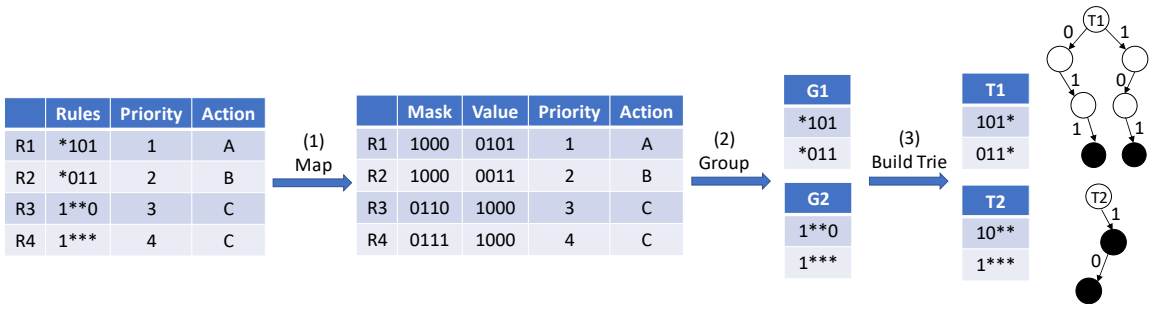


Figure 3.2: GenMatcher framework.

to a minimum to keep performance high while minimizing memory cost. As the full space of all possible groupings with expansion is untenable for any reasonable number of rules, our framework is a heuristic which aims to achieve optimal grouping and expansion without requiring a full search. These phases are detailed in this section.

3.3.1 Map Phase

In map phase, first, we parse rules into four fields, represented by the rule.value, rule.mask, rule.priority, and rule.action fields.

Definition 1. (Rule expression). *Each rule has four properties: value, mask, priority, and action,* as shown in Figure 3.1. These properties have the following definitions:

- **Rule.value:** Represents the non-wildcarded component of the desired match with the wild-card portions set to 0. Generated by parsing all $*$'s (wildcards) in the rule and replacing them with 0s, keeping all other bits unchanged.
- **Rule.mask:** The bit-mask used to clear the wild-card components of the key for comparison against the Rule.value. Generated by parsing all $*$'s and replacing them with 1's, clearing all other bits.
- **Rule.priority:** Where there are multiple rules which match a given key, we choose the rule with the highest priority (1 being the highest). This is provided as input or inferred by the order of rules initially given.

- **Rule.action:** Action to take place as the result of a match, enumerated. e.g., forward the packet to port A, B, or C. Provided as input.

Priority is an important property in matching applications. In packet classification, priority is used to preserve the order semantic of the rules in a rule table, which determines the order of insertion into the trie data structure. The smaller the order value of the rule in the rule table, the higher the priority of the rule. Thus, the rules with higher priority will be inserted earlier, which eliminates the insertion of redundant information. In MPI tag matching, the order semantic of the messages must be preserved. Here, order is represented by the priority property. If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending [44].

After being processed, each rule is now expressed as a set: {value, mask, priority, action}. Given this set, we define the matching process following the operations described in Definition 2.

Definition 2. (Match function).

f(key, mask, value) = (AND(!mask, key) = value).

$$\begin{cases} match = true, & \text{if } AND(!mask, key) = value; \\ match = false, & \text{else.} \end{cases}$$

Example 1. *Assume a rule* $R1$ *is 1**0**11, then its Mask is 01101100, and its value is 10000011. Also assume a key 10001011.*

$10001011 \ \& \ \overline{01101100} = 10001011 \ \& \ 10010011 = 10000011$, *which equals Rule.value.*

$\therefore$ *Key 10001011 matches with* $R1$.

Instead assume the incoming key is 01101011.

$01101011 \ \& \ \overline{01101100} = 01101011 \ \& \ 10010011 = 00000011$, *which does not equal Rule.value.*

$\therefore$ *Key 01101011 does not match with* $R1$.

If multiple rules match, we always choose the one with the highest priority. The match result consists of the priority of the matched rule and its corresponding action. Upon match, the matched rule's action attribute will be returned to the application.

3.3.2 Group Phase

In the *group* phase, we exploit the correlation between rules, observing the similarity of their wildcard patterns, with the goal of transforming all rules into prefixes. Here we have the goal of minimizing the number of groups (since more groups require more search time) while minimizing the amount of rule expansion necessary to construct a trie from each group. Thus before continuing we formally define rule expansion:

Definition 3. (Expand operation). *Expand the non prefix wildcard bits into 1's and 0's.*

Example 2. *Assume rule* $R1$ *is 10*011. Here* $R1$ *is a non-prefix rule.*
After the expansion operation the rule will now be expanded to two new rules without wildcards: 100011 and 101011.
Instead assume rule $R2$ *is 1**0**, where the rightmost *'s form a prefix in common with other rules in the group.*
*After expansion this will result in four rules: 1000**, 1010**, 1100**, and 1110**.*
Thus, the resulting number of expanded rules is determined by the number of non prefixed $*$ *bits. We define the number of* $*$ *in the middle of a rule as* E*. Thus, the number of expanded rules is* 2^E.

Different grouping algorithms generate different resultant groupings. For the rule table shown in Figure 3.2, there are many possible groupings possible; Figure 3.3 illustrates two possible groupings. The first group result is the single group $G1$. The second group result is the two groups $G1'$ and $G2'$. We note, as we will show, building a trie from the single group, $G1$ will require some expansion because a single prefix match is not possible for this set of rules, while building tries for each group $G1'$ and $G2'$ will not require expansion.

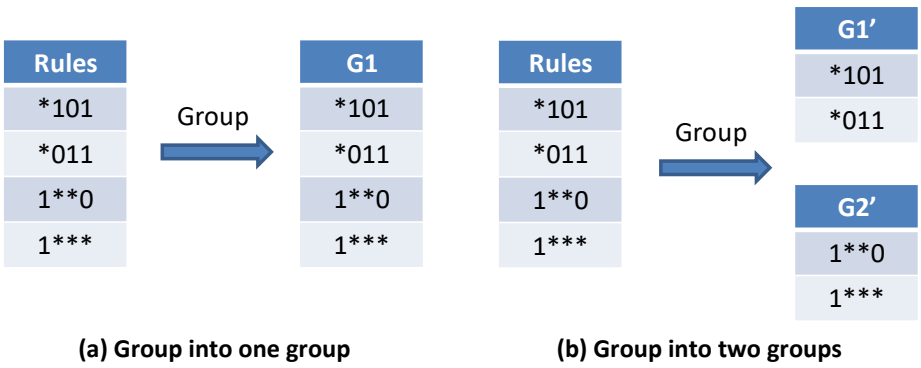


Figure 3.3: Grouping examples.

Since the goal of the group phase is to minimize the number of groups while minimizing the amount of rule expansion necessary to construct a trie for each group, we first attempt to put all rules into one group. If this does not cause the the memory cost to go beyond the given threshold, the group result is the entire rule table; otherwise, we employ our proposed GenMatcher grouping algorithm to split the rules into multiple groups. In the GenMatcher grouping policy, we assign a rule to a group based on the similarity between the rule and the core of the group. Even the rule's similarity is qualified, we need to check the memory cost status of the group if it accepts the new rule. If the memory cost is still within the threshold after the group has absorbed this rule, the rule will be assigned to this group; otherwise, the rule will be assigned to a new group. Thereafter, the number of group is increased.

3.3.3 Build Phase

In the trie build phase, we employ the bit swapping algorithm [10] to rearrange the bits within the rules in each group, such that the wildcards accumulate to the right-most position of the bit string. After swap operations, if some of rules in each group are still not in a prefix format, we need to expand the non-prefix rules into prefix rules.

In Figure 3.4 (a), there is only one group $G1$, generated in the group phase. After the swap operation, three rules, 1*10 , 1*01 and *1**, remain which are not in prefix format. Thus we must expand the three rules into six rules in-order to allow a single trie to be built for this group.

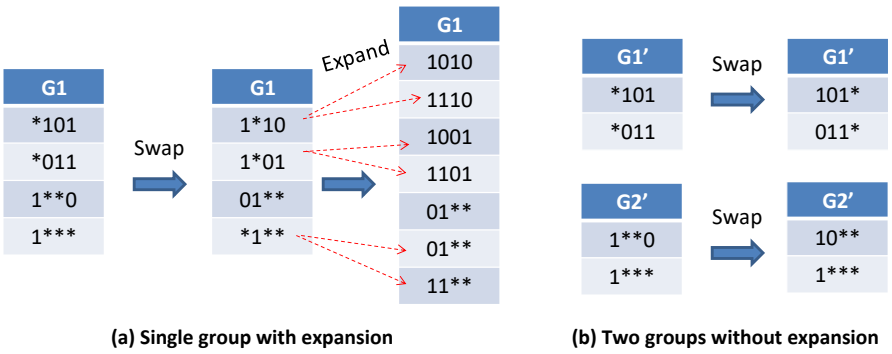


Figure 3.4: Build trie examples.

As shown in Figure 3.4 (b), we must construct two different swap operations to produce prefix matches for each group $G1'$ and $G2'$. After the swap operations, all the rules in each group are in a prefix format. Thus, this second set of groups requires no expansion. As we can observe from Figure 3.3 and Figure 3.4, the groupings chosen in the group phase determines the number of tries and the number of expanded rules.

3.3.4 Objectives and Challenges

As discussed previously, to utilize the efficiency of a trie data structure in performing search operations, each rule must be a prefix match. Thus, rules should be split into groups where each group can be bit-swapped to make the largest possible prefix matches. Since each trie must be traversed sequentially, a larger number of groups will result in a longer time to finish searching than a smaller number[3]. Thus, our strategy is to employ rule expansion to decrease the number of groups, which also tends to reduce the search time. A potential problem with this approach, however, is that the expansion increases the number of trie nodes in the trie data structure, which increases memory utilization. Ultimately, if memory utilization exceeds the system memory limit, the search performance will dramatically suffer.

Objective: Since we must transform arbitrary rules into prefix rules, we may have multiple groups and many rule expansions. Our objective is to minimize the search time while not ex-

[3]The size/depth of each trie itself also impacts search time. Since the complexity of trie search, however, grows as *O(h)* where $h \sim logn$ typically, the number of tries tends to have a greater effect on search time than the size of the trie.

ceeding the memory threshold, defined thus:

- **Search time**: determined by the number of groups and the trie data structure for each group.
- **Memory cost:** determined by the number of trie nodes, which is largely increased through rule expansion.

Challenges: Different grouping results lead to different trie data structures, represented by the number of groups and the number of trie nodes. Thus, we need to resolve the trade-off between the number of groups and the number of trie nodes to obtain an optimal performance balance. In this chapter, we propose the GenMatcher grouping algorithm to form optimized groupings for prefix match searching to achieve high search throughput with a fixed, given memory cost threshold.

3.4 The GenMatcher Grouping Algorithm

In this section, we present the correlation clustering-based (GenMatcher) grouping algorithm in detail.

3.4.1 Similarity Function

Recalling the goal of our application, we want to minimize the search time while not exceeding the memory threshold. In order to minimize the search time, the grouping algorithm needs to generate as few groups as possible, while keeping expansion to a minimum, thus reducing the required memory. We therefore must maximumly group similar rules together with similarity defined as having the most wildcard positions in common.

Since the number of groups and the number of trie nodes are mainly determined by the wildcard distribution in each rule, we can observe the similarity from the *rule.mask* field. Considering two rules R_1 and R_2, we define the similarity function $s(R_1, R_2)$ as follows.

Definition 4. The similarity between rule R_1 and R_2 is collected from their wildcard distribution,

as defined in Eq. 3.1.

$$s(R_1, R_2) = \begin{cases} M(R_1.mask \ \& \ R_2.mask), & \text{if } R_1.mask \neq 0 \wedge R_2.mask \neq 0 \\ 1, & \text{if } R_1.mask = 0 \parallel R_2.mask = 0 \end{cases} \tag{3.1}$$

where $M(\cdot)$ counts the number of 1's set in a bitstring.

Since expansion is determined by the wildcard distribution, the intersection of similarities among rules with non-zero mask is the key factor in grouping. If $R_1.mask \neq 0 \ \wedge \ R_2.mask \neq 0$, the similarity equals to the number of $*$ that appear on both R_1 and R_2; If $R_1.mask = 0 \parallel R_2.mask = 0$, which means that we always can group these two rules together since there is no expansion. Because rules which have mask = 0 (*i.e.* no wildcards) cause no rule expansion and can be placed equally in any group, so we set the similarity to 1[4] when $R_1.mask = 0 \parallel R_2.mask = 0$. If the rule is 64-bit, the similarity belongs to $[0, 64]$, where 0 denotes no similarity between two rules.

3.4.2 GenMatcher Grouping Algorithm

We now present our correlation clustering-based GenMatcher grouping algorithm for the group phase in GenMatcher. Given a set $\mathcal{D}$ of rules, our GenMatcher algorithm generates a set $\mathcal{O}$ of groups. It requires a parameter θ, the given memory threshold.

The pseudocode of GenMatcher grouping is shown in Algorithm 1. The algorithm consists of two steps. In the first step (lines 2-10), the *caculateMatrix* function computes the similarity of any two rules in the rule set $\mathcal{D} = \{R_1, R_2, \cdots, R_N\}$ utilizing Definition 4. In the second step (lines 12-27), the *assignCluster* function assigns rules into groups according to the similarity matrix $S_{N \times N}$, calculated by the *caculateMatrix* function.

In the second step, the rule that has the most correlation with other rules in the rule set is

[4]In our GenMatcher grouping policy, if the similarity between a rule and the core of a group is 1, the rule will be assigned to a group if it does not break the memory limitation. We arbitrarily choose the value 1 because it is small enough that does not affect the selection of the core of a group, which we want to have the greatest summation of similarities between itself and any other rules.

Algorithm 1 GenMatcher Grouping Algorithm.

Input: $\mathcal{D} = \{R_1, R_2, \cdots, R_N\}$
Output: $\mathcal{O} = \{G_1, G_2, \cdots, G_{num}\}$

Set θ = 64MB
function CALCULATEMATRIX($\mathcal{D}$)
    $S_{N\times N} \leftarrow \emptyset, L_{N\times 1} \leftarrow \emptyset$
    for i=1; i $\leq$ N; i++ **do**
        for j=1; j$\leq$ N; j++ **do**
            $S_{ij} \leftarrow s(R_i, R_j)$
            $L_i \leftarrow L_i + S_{ij}$
        end for
    end for
end function

function ASSIGNCLUSTER($\mathcal{D}, S_{N\times N}, L_{N\times 1}$)
    $index \leftarrow \underset{i\in\mathcal{D}}{\operatorname{argmax}}(L_{N\times 1})$
    for i=1; i $\leq \mathcal{D}.size()$; i++ **do**
        if $S_{index,i} > 0 \wedge$ total memory cost $< \theta$ **then**
            $H[1] \leftarrow R_i$
        else
            $H[2] \leftarrow R_i$
        end if
    end for
    $\mathcal{O} \leftarrow H[1]$ /* a new group is generated */
    if $H[2].size() == 0$ **then**
        return $\mathcal{O}$
    else
        $assignCluster(H[2], S_{N\times N}, L_{N\times 1})$
    end if
end function

chosen as the core of a new group. The procedure is shown in line 13. The *index* is the index of the core of the new group. If the similarity between a rule and the core is positive and under the given memory limit, the rule is added to the vector $H[1]$; otherwise, it is added to the vector $H[2]$. $H[1]$ is a new group of the output $\mathcal{O}$. $H[2]$ is a vector storing all the rules not qualified in $H[1]$. $H[2]$ will be processed in function *assignCluster* until empty. After we obtain the grouping result $\mathcal{O}$, we can build the trie data structure, as shown in Figure 3.2. Thereafter, we process the binary search operations for all the incoming keys.

We now present, in Lemma 1, the time complexity of the GenMatcher algorithm. In Lemma 2, we examine the determinant factors of search performance for our GenMatcher algorithm.

Lemma 1. The time complexity of our GenMatcher grouping algorithm is $O(N^2)$, where N is the number of rules.

Proof. For the function *calculateMatrix* (lines 2-10), the time complexity is $O(N^2)$. For the function *assignCluster* (lines 12-27), the time complexity is $O(N * num)$, where num is the number of groups. Thus, the time complexity of the GenMatcher algorithm is dominated by the function *calculateMatrix*, which is $O(N^2)$. □

Lemma 2. For our GenMatcher grouping algorithm, the search performance is primarily determined by the number of groups; the fewer groups, the better the search performance. When the number of groups is fixed, fewer trie nodes always performs better.

Proof. Let us compare two different groupings constructed using an identical set of rules. Assuming that each trie generated by the two groupings are balanced, the performance will not impacted by bias keys. The first grouping results in N trie nodes contained within a single group. The second grouping results in M trie nodes across two groups; one group has M_1 trie nodes, the other one has M_2 trie nodes. The average search attempts[5] for the first grouping is $\log_2(N)$. Since we assume tries must be searched sequentially to get the matched result, the average search attempts for the second grouping is $\log_2(M_1) + \log_2(M_2)$, where $M_1 + M_2 = M$. Thus, the search

[5]The number of search attempts for a given key is the depth that a key needs to traverse in a trie data structure before a match can be determined.

performance of a single group performs better than two groups when

$$\log_2(N) < \log_2(M_1) + \log_2(M_2) = \log_2(M_1 M_2)$$

$$\implies N < M_1 M_2$$

If $M_1 = M_2$, $\log_2(N) < 2\log_2(\frac{M}{2}) \implies N < \frac{M^2}{4}$. If the number of groups is G, then the statement is true when $N < M_1 M_2 \cdots M_G$. As G increases, $M_1 M_2 \cdots M_G$ is more than likely larger than N, especially when the G groups have an even number of trie nodes, that is, $N < \frac{M^G}{G^G}$. In general, $M \gg G$. Therefore, as G increases, M^G will always grow faster than G^G; thus, we can say that the search performance is primarily determined by the number of groups.

Let us now consider two groupings with an identical group size, generated by the same rule set. If the keys are unbiased, the grouping result with the least number of trie nodes has fewer search attempts. As the search time is determined by the number of search attempts, the search time is proportional to the number of trie nodes. If the keys are biased, a grouping with more trie nodes might perform better when the trie data structure is unbalanced. As needed, we can employ tree balance techniques [45, 46, 47, 48, 49, 50] to rebalance the data structure. Thus, when the number of group is fixed, fewer trie nodes always performs better. □

3.5 Evaluation

In this section, we evaluate $GenMatcher$ on rule sets generated from packet capture (PCAP) traces. We first present the evaluation methodology, followed by the correctness and scalability of the $GenMatcher$ grouping algorithm. Finally, we compare the results obtained versus previous techniques.

3.5.1 Methodology

We program $GenMatcher$ framework in C++, leveraging our in-house developed binary trie data structure and the linear vector data structure in C++ standard library (STL). A rule generation heuristic developed by McHale *et. al* [51] was used to synthesize a set of rules relevant

to a given packet capture (PCAP) trace. Rules used in this project consisted of IPv4 source and destination addresses, resulting in a key width of 64-bits. Keys used to measure matching performance were constructed using the source and destination IPv4 address of every packet extracted from the given PCAP trace. The PCAP dataset (equinix-sanjose.dirA.20120119-125903) examined in this chapter was taken from an internet backbone link and provided by CAIDA [52]. CAIDA's anonymization preserves relative flows across packets in a trace. These traces provide a more realistic dataset with less entropy compared to randomly generated rules and keys.

For the rule tables, we use our generator to generate 11,000 rules based on the CAIDA PCAP data [52]. We randomly pick 10 different sets of rule samples with 6 different rule table sizes (256, 512, 1024, 2048, 4096, 8192). We extract 28,744,877 keys from the traces as previously described. We set a memory cost threshold of 64 MB. We evaluate our *GenMatcher* on a real system with 512 GB DRAM and Intel Xeon E5-2697A V4 32-core 2.6GHz processor. L1d cache is 32k, L1i cache is 32k, L2 cache is 256k, and L3 cache is 40960k. All tests are executed with one single thread.

We evaluate the performance in terms of search time, configuration time, and memory cost. First, in order to show the correctness and scalability of the *GenMatcher* grouping algorithm, we compare *GenMatcher* against a brute force group search algorithm. For the brute force algorithm, the objective is to find the optimal group results which results in the minimal number of groups with minimal number of trie nodes while not exceeding the memory threshold. The time complexity of the brute force algorithm is $O(N^N)$, where N is the number of rules.

Second, we compare *GenMatcher* against two arbitrary matching algorithms: the Linear arbitrary matching algorithm, which serves as a baseline, and a software-only version of the Bitweaving [10] algorithm. Linear arbitrary matching employs C++ STL vectors to build a rule table and search the rules sequentially. The goal of Bitweaving is the compression and reformatting of rules such that they could be best implemented in a TCAM. Their proposed technique is composed of a bit swapping algorithm and a bit merging algorithm. In the bit swapping stage, they employ a grouping policy. First, they sort the rules by the wildcard positions and the number of wildcards in each rule. After sorting, the rules with similar wildcard distribution are moved next to each

other. Second, they start grouping rules from the beginning of the sorted rule list. They define a cross-free condition to determine if the adjacent rules can be grouped together. If the rules' wildcard distribution has intersection and all the wildcards can be swapped to the right side of the rules by applying a same permutation pattern, the rules can be grouped together; otherwise, they will make another group. The grouping process will not stop until the last rule in the rule list has been examined. Within each group, a single permutation is applied to each rule's bits to produce a reordered rule. Bitweaving employs its grouping policy to transform all the non-prefix rules into prefix rules. After grouping, all the rules in each group are in a prefix format.

3.5.2 Comparison with Brute Force Grouping

In Lemma 2, we prove that the search performance is determined by the number of groups and trie nodes. Thus, if the **GenMatcher** algorithm generates the same groups and number of trie nodes compared with the brute force (**BF**) algorithm, the **GenMatcher** algorithm is achieving the optimal grouping. Since the brute force algorithm is an NP-complete problem, we can only evaluate it on small rule samples due to the runtime limitation. We randomly choose 10 different sets of rule samples ranging from 10 rules to 20 rules from the overall rule set.

Table 3.1: The number of trie nodes for BF and GenMatcher.

$Rule\#$	10	11	12	13	14	15	16	17	18	19	20
BF	338	435	452	464	535	587	666	708	671	724	734
Gen	338	435	452	464	535	587	666	708	671	724	734

Table 3.1 and Table 3.2 shows the number of trie nodes and the number of groups generated by the **GenMatcher** algorithm and the brute force algorithm on selected rule samples [6]. As shown in Table 3.1 and Table 3.2, we observe that **GenMatcher** and **BF** generate the same number of trie

[6]Since the rules were chosen randomly, sometimes a larger set of rules produces a slightly larger prefix (and smaller trie), than a smaller set of rules. Thus, 18 rules has less nodes than 17.

Table 3.2: The number of groups for BF and GenMatcher.

$Rule\#$	10	11	12	13	14	15	16	17	18	19	20
BF	2	2	2	2	2	2	2	2	2	2	2
Gen	2	2	2	2	2	2	2	2	2	2	2

nodes (**Gen** represents **GenMatcher**). In each case the same number of groups were generated as well. Since the search performance is determined by the number of trie nodes, the search performance of the two algorithms are the same.

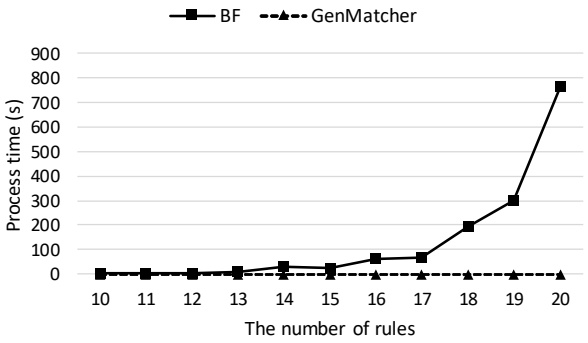


Figure 3.5: Time complexity comparison.

With respect to the time complexity, the **BF** is much worse than **GenMatcher** algorithm. The average process time for grouping 10 to 20 rules is shown in Figure 3.5. The process time for **GenMatcher** is ineligible at only 1 μs since the size of the rule samples is very small. However, the process time for **BF** increases exponentially with the number of rules. The process time of generating the group results for **BF** is nearly 13 minutes when the rule sample size $= 20$.

Overall, the performance of our proposed **GenMatcher** algorithm for this range of rule counts matches optimal. Further, the time complexity of **GenMatcher** is much better than the **BF** algorithm.

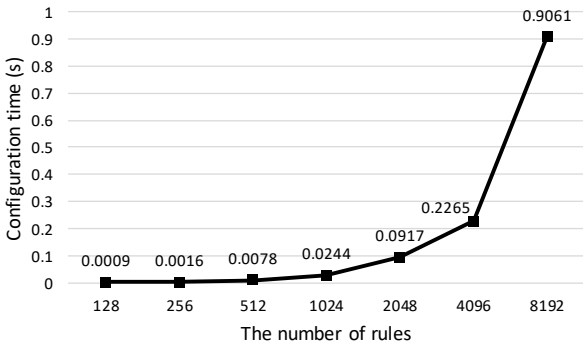


Figure 3.6: GenMatcher grouping algorithm scalability.

3.5.3 Scalability

To show the scalability of **GenMatcher**, we evaluate its configuration time by scaling up the number of rules from 128 to 8192. Figure 3.6 shows the configuration time (*i.e.* the time to generate all tries) of **GenMatcher** when scaling the number of rules. The configuration time consists of two parts. One is the time for processing **GenMatcher** to get the group results. The other one is to build the trie data structure based on the group results. Thus, the configuration time is a summation of the process time and the build time, where the build time is the summation of rule rearrange time and rule insertion time, which are happened during build trie phase. The rule rearrange time is the time cost during bit swapping operation. The rule insertion time is the time cost for inserting all the rules into the trie data structure. As expected, configuration time increases with the number of rules. However, for a 8192 rules, the configuration time of **GenMatcher** is less than one second. Thus, **GenMatcher** incurs little overhead for a large set of rules.

Since **Bitweaving** and **GenMatcher** both have group phase and build trie phase, their configuration time elements are the same. Figure 3.7 shows the configuration time comparisons in terms of build time and process time with respect to **Bitweaving** and **GenMatcher**. **Bit** represents **Bitweaving**, and **Gen** represents **GenMatcher**. Here we see that **Bit_build** and **Gen_build** both take less time than **Bit_process** and **Gen_process**. On the 8192 case, we observe that **Gen_build** is larger than **Bit_build**. This is because **GenMatcher** generates less groups than

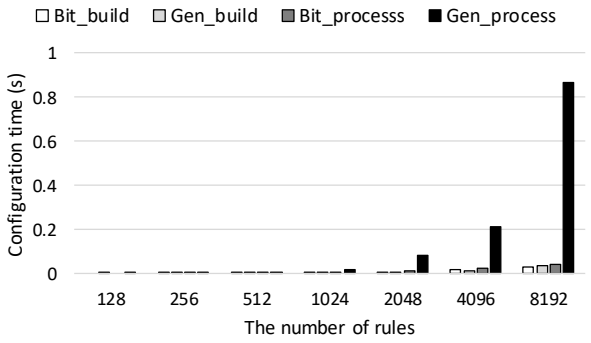


Figure 3.7: Configuration time comparisons between Bitweaving and GenMatcher.

Bitweaving by trading off memory for performance. The insertion time of **GenMatcher** is larger than **Bitweaving** due to the rule expansion. For the process time comparison, since the time complexity of **Bitweaving**'s grouping is $O(N)$, which is smaller than $O(N^2)$, **Bit_process** is smaller than **Gen_process**. Although the configuration time of **GenMatcher** is larger than **Bitweaving**, the resulting search performance is much better than **Bitweaving**, which is the goal of our *GenMatcher* framework. The search performance is described in Section 3.5.4.

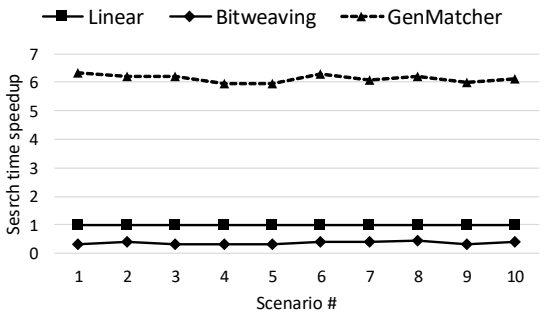
3.5.4 Performance Comparisons

We compare **GenMatcher** against **Linear** and **Bitweaving** in terms of search time and memory cost.

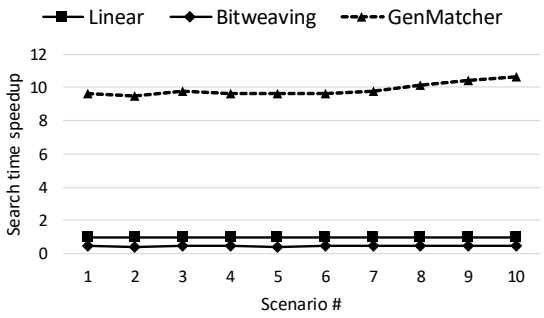
3.5.4.1 Search time:

Figure 3.8 shows the search time speedup with respect to **Linear**, **Bitweaving** and **GenMatcher**. For all the 6 different rule sizes (256, 512, 1024, 2048, 4096, 8192), **GenMatcher** achieves the best performance. In the figure, for each rule size, performance of 10 different scenarios (randomly chosen rules) are shown.

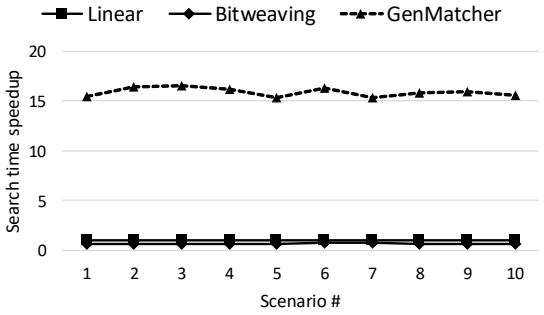
In the figure we see that **GenMatcher** generally provides greater speedups for larger numbers of rules. This is because the performance of **Linear** degrades (linearly) with increasing numbers of rules, while the performance of both **GenMatcher** and **Bitweaving** tend to be dominated by the number of groups each technique creates.



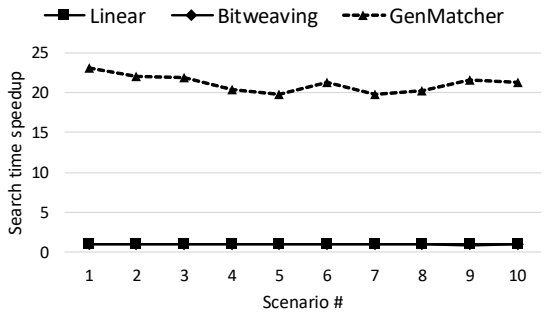
(a) Rule num = 256.



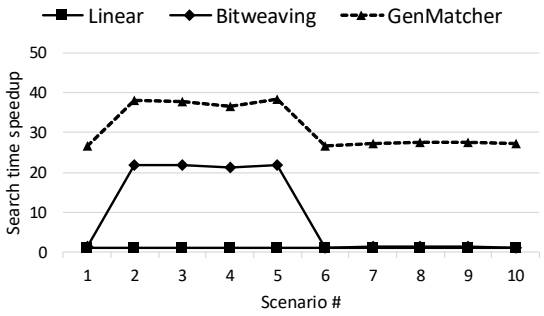
(b) Rule num = 512.



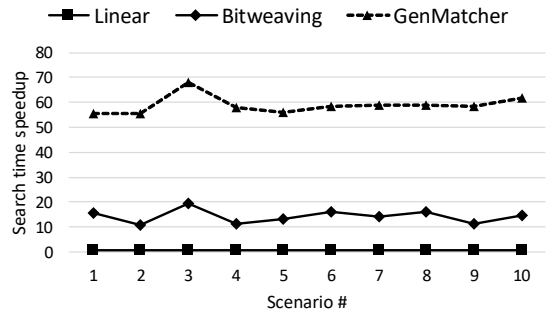
(c) Rule num = 1024.



(d) Rule num = 2048.



(e) Rule num = 4096.



(f) Rule num = 8192.

Figure 3.8: Search time speedup comparison, normalized against Linear.

Between **Bitweaving** and **GenMatcher**, since the search time is determined by the number of groups and the number of trie nodes within the given memory threshold, the performance is always better with fewer groups. Because **Bitweaving** generates more groups than **GenMatcher**, **GenMatcher** always outperforms **Bitweaving**.

The $Rule\ num = 4096$ case, shown in Figure 3.8 (e), shows an interesting behavior. In the figure we see that both **Bitweaving** and **GenMatcher** produce substantially better results for

Table 3.3: Grouping result on $Rule\ num = 4096$

$S\#$	1	2	3	4	5	6	7	8	9	10
Gen	2	1	1	1	1	2	2	2	2	2
Bit	62	1	1	1	1	62	62	62	60	62

Table 3.4: The number of trie nodes result on $Rule\ num = 4096$

$S\#$	2	3	4	5
Gen	43163	42284	42806	42839
Bit	48522	47414	47610	48065

scenarios 2-5 than for the others. Table 3.3 shows the number of groups generated for each scenario by the two techniques. Here we see that **GenMatcher** and **Bitweaving** both generate only one group for scenarios 2-5. Despite having the same number of groups, **GenMatcher** still produces better performance. This is because **GenMatcher** generates a smaller number of trie nodes than **Bitweaving**, which is shown in Table 3.4. The different number of trie nodes is obtained by the different number of inserted rules in trie data structure, as shown in Table 3.5. In our **GenMatcher**, we insert rules by its priority order. In **Bitweaving**, it sorts the rules by an ascending order of the number of wildcard in a rule [10], which breaks the rules' priority order. Thus, **Bitweaving** does not insert rules by its priority order, which result in redundant trie nodes in trie data structure.

Since **Bitweaving** typically generates a large number of groups to maintain the given memory threshold, its performance is even worse than **Linear** when the number of rules is small. However, the performance of **Bitweaving** improves with increasing rules. When the number of rules is up to 4096, as shown in Figure 3.8 (e), **Bitweaving** surpass **Linear**.

Out of the 6 different rule counts, **GenMatcher** achieves the best performance on $Rule\,num = 8192$. On average, **GenMatcher** achieves an 58.9X speedup compared to the baseline. To sum-

Table 3.5: The number of inserted rules result on $Rule\ num = 4096$

$S\#$	2	3	4	5
Gen	1913	1923	1908	1926
Bit	3125	3197	3156	3102

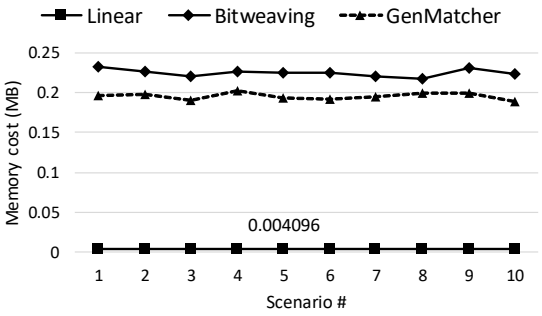
marize, the best search performance can be achieved by minimizing the number of groups while remaining under the memory cost threshold.

3.5.4.2 Memory cost:

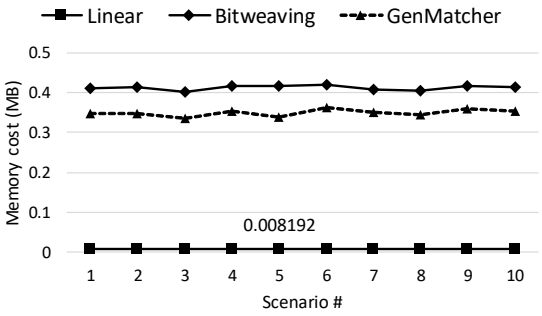
Figure 3.9 shows the memory cost with respect to 10 different scenario # on various number of rules. Out of the 6 different rule counts, **Linear** has the least memory cost, which is determined by the number of rules. The memory cost of **Linear** is the product of the number of rules and the size of a rule. The size of a rule consists of two 64-bit words for the *mask* field and *value* field. Thus, the size of a rule is $16B$. Both **Bitweaving** and **GenMatcher** are based on trie data structures, thus the cost is determined by the number of trie nodes. Their memory cost is the product of the number of trie node and the node size. The trie node consists of two pointers (each pointer is 64 bit) and one integer (32 bit). The node size is $8 + 8 + 4 = 20B$.

Between **Bitweaving** and **GenMatcher**, without rule expansion, the memory cost is consistent, depending on the number of groups. The fewer number of groups, the smaller the memory cost. However, the memory cost is more diverse when rule expansion occurs. In Figure 3.9 (a, b, c, d), **Bitweaving** always consumes more memory than **GenMatcher**. This is because the number of groups generated by **Bitweaving** is larger than that generated by **GenMatcher**, and there is no rule expansion for both of them.

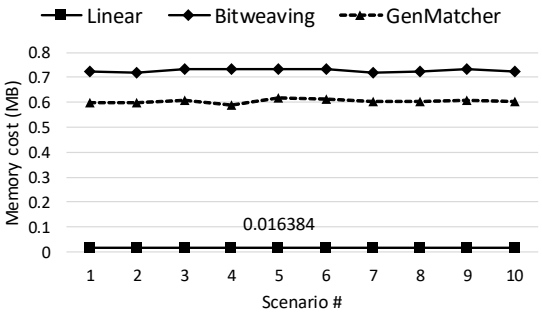
We take the $Rule\ num = 4096$ and $Rule\ num = 8192$ cases as examples. Table 3.3 shows the grouping results for $Rule\ num = 4096$. The performance is consistent with the number of groups. Note that on scenarios 2, 3, 4, and 5, the cost of **Bitweaving** and **GenMatcher** are lowest because the number of the groups are the smallest and there is no rule expansion.



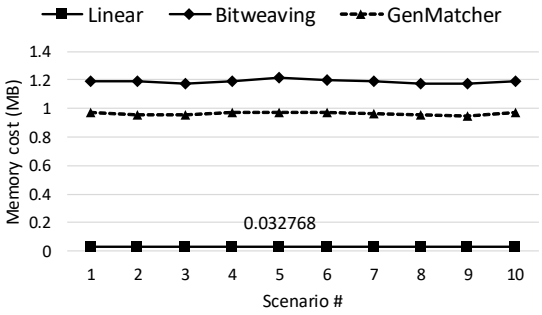
(a) Rule num = 256.



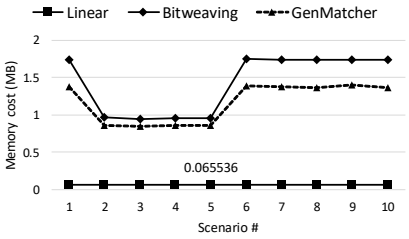
(b) Rule num = 512.



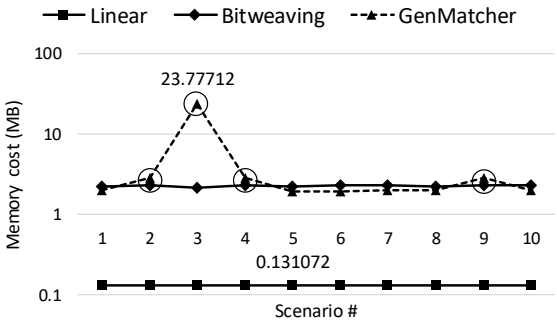
(c) Rule num = 1024.



(d) Rule num = 2048.



(e) Rule num = 4096.



(f) Rule num = 8192.

Figure 3.9: Memory cost versus the number of groups at different rule sample sizes.

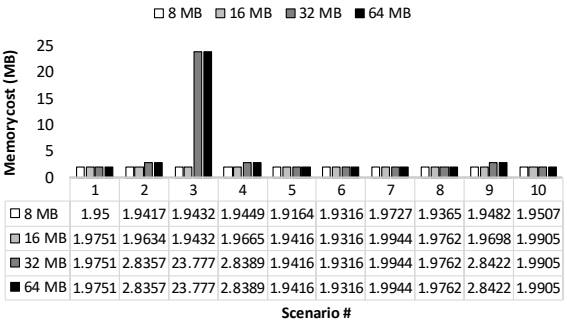
Table 3.6 and Table 3.7 shows the grouping result and expansion result for $Rule\ num = 8192$, respectively. Note that, since there is a large number of expanded rules on scenario 2, 3, 4 and 9, though **GenMatcher** generates less groups than **Bitweaving**, **GenMatcher** requires much more memory. As shown in Figure 3.9 (f), on scenario 2, 3, 4 and 9, **GenMatcher** has a greater memory cost than **Bitweaving**. To achieve the best search performance, we trade-off memory cost. This meets our objective: minimize the search time while not exceeding the memory threshold.

Table 3.6: Grouping result on $Rule\ num = 8192$

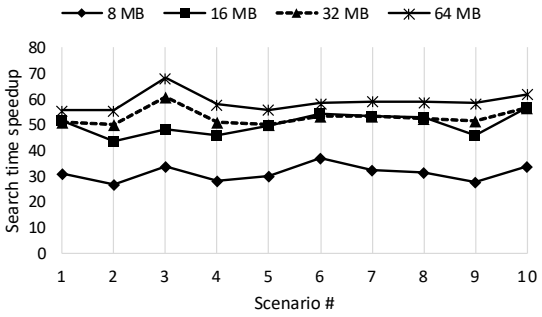
$S\#$	1	2	3	4	5	6	7	8	9	10
Gen	2	2	1	2	2	2	2	2	2	2
Bit	5	9	3	9	7	5	7	5	9	7

Table 3.7: Expansion result on $Rule\ num = 8192$

$S\#$	1	2	3	4	5	6	7	8	9	10
Gen	51360	55088256	536887296	55088256	51360	0	12880	51360	55088256	51360
Bit	0	0	0	0	0	0	0	0	0	0



(a) Memory cost.



(b) Search time speedup.

Figure 3.10: Performance comparisons at different memory threshold.

Figure 3.10 shows the performance of GenMatcher at different memory thresholds (8 MB, 16 MB, 32 MB, 64 MB). For all 10 scenarios examined, GenMatcher only produces a significantly different memory cost for the third text. In this test, GenMatcher chooses to create fewer groups when more memory is available, through the more aggressive use of rule expansion. These fewer groups lead to a significant performance improvement as shown in Figure 3.10 (b), where thresholds of 32 MB and 64 MB perform significantly better. We note that, GenMatcher is highly conservative when configured to maintain a given memory threshold, thus the actual memory used

is often much lower than the threshold. This is because GenMatcher uses a heuristic at grouping phase to estimate the upper bound of memory which might be used during the expansion phase. This heuristic often leaves some memory on the table but is much faster (at configuration time) than iteratively grouping and expanding to hit a given memory target.

Figure 3.11 shows the memory cost per rule with respect to Bitweaving and GenMatcher. For all the 6 different rule sizes (256, 512, 1024, 2048, 4096, 8192). For all cases except 8192, GenMatcher has a lower memory cost per rule than Bitweaving. In the 8192 case, to preserve matching performance by reducing groups, GenMatcher expands more rules than Bitweaving (while remaining under the defined memory budget), thus, GenMatcher uses more memory per rule than Bitweaving. However, GenMatcher maintains a better search time in return for the extra memory per rule.

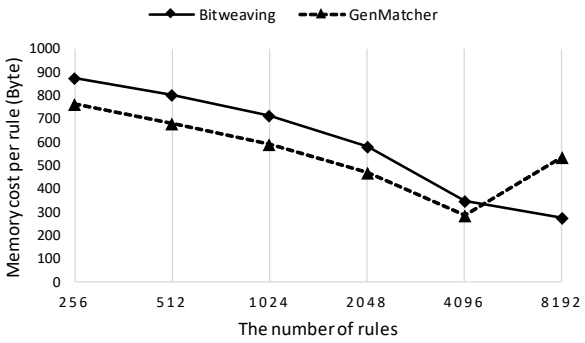


Figure 3.11: Memory cost per rule.

3.6 Conclusions

This chapter proposes *GenMatcher*, a generic, arbitrary, software-only matching mechanism for fast, efficient, searches under a given memory threshold. *GenMatcher* employs our proposed grouping algorithm to assign the arbitrary rules with the greatest similarities into the same group. *GenMatcher* generates a minimal number of groups within a memory threshold, and is able to build an efficient trie data structures to perform fast binary searching. For a rule table with a size

8192, *GenMatcher* achieves a mean speedup of 58.9X over a Linear baseline.

4. GenSMatcher: A GENERIC SIMD-BASED ARBITRARY MATCHING FRAMEWORK

This chapter presents a generic SIMD-based arbitrary matching mechanism that further improves the matching performance. First, we introduce a background of a height-optimized trie data structure. Second, the motivation of our GenSMatcher is presented. Then, the design section describes a detailed implementation of GenSMatcher. Finally, the evaluation section compares our SIMD-based arbitrary matching mechanism with our previous proposed GenMatcher in search performance, memory cost, and insertion time.

4.1 Introduction

Packet classification is the critical component in a switch network, which classifies/forwards internet packets to flows according to the pre-defined rules generated by the different network applications and their configurations [53]. With the advent of SDN technology, the applications have become more flexible since the control plane is decoupled from the data plane. This separation increases the demand for software-based generic packet classification methods. In traditional switch networks, rules consist of five fields: source IP address, destination IP address, source port number, destination port number, and protocol [54]. However, in an SDN switch network, the rules have more fields as defined in OpenFlow [55], a de facto standard of SDN. OpenFlow defines the communication mechanism and message format between the control plane and the data plane. The latest OpenFlow ver. 1.5 defines 45 different flow match fields, in which about half of them involve wildcards [56]. Therefore, the multi-field packet classification problem becomes more complex and challenging [57, 40, 38].

While there exists plenty of research on packet classification, most of them focus on prefix and range matching [9]. With the prevalence of High-Performance Computing (HPC) applications, the software-defined network technique will be deployed in HPC networks [58], requiring high-performance packet classification [59]. Different applications generate various ordering of multi-fields, resulting in wildcard rules with diversified wildcard bits. Such a variety of wildcard bits

renders multi-field packet classification a challenging function for SDN devices. There are four different matching types for packet classification: exact match, prefix match, range match, and arbitrary match [3]. Prefix match and arbitrary match both contain wildcard bits. The wildcard bits of prefix match is located at the end of a rule, whereas arbitrary matches may have wildcard bits at any bit position. Since all matching types can be represented in arbitrary matches, the essential task is to improve the generic arbitrary matching performance.

There are few prior works on arbitrary matching. Meiners et al. proposed Bitweaving [10], a hardware-based compress approach to reduce the number of ternary content-addressable memory (TCAM) rules. Bitweaving utilizes the TCAM, which is expensive, power-hungry, and capacity-limited. Therefore, the Bitweaving method is not suitable for large data applications. We introduced the GenMatcher [3], a software-based generic clustering-based arbitrary matching framework, which is implemented with a binary trie data structure. Compared to the linear search and bitweaving [10] method, GenMatcher gained a significant speedup on search performance. Although GenMatcher improved the arbitrary matching performance, it does not leverage the modern processor features, such as SIMD instruction. Besides, GenMatcher cannot keep up with the scalable performance with the sizeable blooming scale of input since the binary-trie data structure cannot satisfy the high-performance matching needs. Ultimately, with the demand for high-performance matching from big data and HPC applications, GenMatcher restricts the room for improvement in search performance and memory cost. Thus, the high-performance arbitrary matching is still an open problem.

Binna et al. proposed the height optimized trie (HOT) data structure, which takes advantage of the SIMD instruction and modern cache features that exploit data localities to accelerate the searches and reduce memory usage. Unfortunately, HOT can not be directly applied for arbitrary matching because HOT does not support wildcard rules. In addition, HOT is incapable of prioritizing results. Since wildcard matching can have multiple matching results, the priority feature is needed to choose the correct candidate, usually with the highest priority. Due to this, there is no existed work related to the modern processor feature supported arbitrary matching. In order to

fill out this gap, this chapter aims to achieve a high-performance search throughput and memory cost by utilizing the modern processor features for arbitrary matching.

A key to achieve high matching performance is to ensure that the matching data structure can utilize the modern CPUs' advanced features. This chapter proposes *GenSMatcher*, a generic SIMD-based arbitrary matching framework, which utilizes the SIMD modern feature to improve the matching performance in search time and memory cost. Our proposed GenSMatcher employs the Height Optimized trie (HOT) data structure presented by Robert Binna et al. [25], in which the trie node is composed of a fixed upper bound of binary trie node. The trie node has a data-dependent span and a fixed maximum fanout, enabling a consistently high fanout for arbitrary key distributions and efficient search using the SIMD feature.

To process arbitrary matching, we propose a novel mechanism to deal with the wildcard rules. First, we defined wildcard rule representation [3]. After the interpretation of wildcard rules, we can insert them into the HOT data structure. Second, HOT only carries partial bits of a rule, represented by the discriminative bits, which efficiently reduces the memory cost. Since the wildcard rules have $*$, which can be either 1 or 0, we propose a novel algorithm to extract the effective partial bits based on the rules' distribution. Third, to guarantee the integrity of wildcard rules' information, we must store the rules' complete bits in the corresponding leaf nodes.

Wildcard rule representation: We design a rule representation method that can interpret wildcard rules into three fields. The representation allows us to insert the wildcard rules into the trie data structure.

Challenges: HOT is able to represent a rule with only partial bits, which reduces the memory footprint. However, we cannot use its discriminative bit selection method to choose the partial bits for a wildcard rule. To enable wildcard matching operation, we design a GenSMatcher extraction algorithm to obtain a wildcard rule's partial bits to guarantee the matching outcome. As wildcard rules can cause multiple matching rules for an incoming key, we must choose the highest priority to get the correct matching result. Therefore, we need to use some extra space for the leaf node to store the wildcard rules. Our goal is to utilize the modern processor features

to improve the arbitrary matching performance. To implement the efficient arbitrary matching, we must address these challenging technical problems:

- We must modify the HOT data structure to support wildcard matching.
- We need to develop a partial bits extraction algorithm to choose the affected discriminative bits from the corresponding node's discriminative bits set.
- We must guarantee that the arbitrary matching result is correct.

Contributions: $GenSMatcher$ is an advanced arbitrary matching framework that can take advantage of modern processor's features. The contributions are:

- $GenSMatcher$ is an enhanced SIMD-based arbitrary matching framework, processing any type of matching.
- We develop a novel GenSMatcher extraction algorithm to extract the effective partial bits of a wildcard rule.
- Experiments show that GenSMatcher achieves search time speedup by utilizing the SIMD feature on average by 2.7X compared to GenMatcher, and up to 6.17X reduction for the memory footprint.

4.2 Background

Arbitrary matching can be implemented either by hardware-based or software-based approaches. TCAM [29] is a hardware-based solution which can process the arbitrary rules in parallel. Nonetheless the TCAM is power-hungry and expensive, and it cannot scale with the number of rules [28], especially for the big data and HPC applications. This dissertation will focus on the software-based approach. The goal is to develop a data structure that is efficient for high-performance arbitrary matching.

We have witnessed the rapid evolution of processor architecture and memory system, whose new features have been proven beneficial to arbitrary matching [22, 23, 60, 26]. Meanwhile,

trie is a lexicographic data structure where nodes can share their common prefix. Even with the increasing size of data sets, the trie height is independent of the rule size; instead, it is determined by the length of the rule and the node design. In particular, traditional tries cannot be directly used in arbitrary matching where wildcards may appear at any bit positions [3]. However, the trie can be applied for the arbitrary rules via certain transformations. Our previous work proposed a generic arbitrary matching framework, GenMatcher. GenMatcher can convert the arbitrary rules into prefix rules inserted into a binary trie data structure. The search performance of a binary trie depends on the trie depth, which equals the length of a rule. Hence we need to decrease the trie height in order to improve the search performance. In this chapter, we will concentrate on optimizing the trie data structure to achieve high-performance arbitrary matching.

4.2.1 Relationship with Prior Art

Table 4.1 shows the complexity comparisons between all the following trie data structures, where W is the key length, M is the span size, and T[1] is the trie node size. The space complexity in Table 4.1 are based on the worst case.

GenMatcher [3] groups rules and converts wildcard rules into prefix rules, which are later inserted into a binary trie data structure. For a 64-bit integer rule set, the worst trie height will be 64 if there are no wildcard rules. During the search process, we have to search each bit sequentially to get a matching result. The trie heights should be reduced for optimized performance. Therefore, we need to optimize the trie data structure from cache efficiency and modern processor utilization to reduce the height.

We explore the Patricia trie (or radix tree) [24], a compressed version of a trie. In a binary trie, for a 64-bit integer, each trie node can have up to two edges: bit-0 and bit-1. Each non-wildcard bit will be stored in the trie as a trie edge, while in a Patricia trie, multiple bits can be stored in

[1]The trie node size of each trie data structure in Table 4.1 is unique, determined by its trie node layout.

[2]Here the binary trie is a complete tree and each trie node has two pointers.

[3]Patricia trie is a path-compressed trie whose height might be reduced based on the data distribution. On average, $W' \leq W$.

[4]Since the ART trie node can have different span sizes, M' represents its average span size.

[5]W'' is determined by the data distribution, and $W'' < W'$.

Table 4.1: Complexity comparisons of the different trie data structure

	Trie height	Space complexity	SIMD feature
Binary trie	$O(W)$	$O(2^W \cdot T)^2$	No
Patricia trie	$O(W')^3$	$O(2^{W'} \cdot T)$	No
M-ary trie	$O(W/M)$	$O((2^M)^{(W/M)} \cdot T)$	Yes
ART trie	$O(W/M')^4$	$O((2^{M'})^{(W/M')} \cdot T)$	Yes
HOT trie	$O(W'')^5$	$O(2^{W''} \cdot T)$	Yes

the Patricia trie as a single edge. Thus, multiple binary nodes can be compressed into one node, which reduces the trie height and decreases the number of nodes resulting in saving memory.

To further decrease the trie height, we study the M-ary trie, whose span is M. In contrast to binary trie, M-ary trie increases the span from 1-bit to M-bit. The maximum number of children of each trie node is 2^M. Since modern CPUs allow multiple comparisons to be performed with a single SIMD instruction, M-ary reduces the search time with a smaller height. Consequently, the number of cache misses is reduced because 2^M comparisons can be performed for each cache line loaded from main memory [26]. Hence, compared with the binary trie, M-ary trie is not only more effective in searching by utilizing SIMD instruction, but it is also cache-friendly. Furthermore, the trie height is reduced from W to W/M, where W is the key length. However, as the span size increases, the memory cost increases exponentially because each node is allocated 2^M pointers in an array. Nowadays, some big data applications are more sparsely distributed, which causes a large amount of memory wasted. Thus, for different applications, we need to adopt varying span sizes to save memory costs. Therefore, we explore the adaptive radix tree [26] (ART) data structure next.

While a traditional radix tree requires the trade-off of tree height with memory cost by setting

a globally valid span parameter, ART proposes a node represented with a varying span size. The number of child nodes determines the size. If a node has many child nodes, ART assigns an enormous span to this node; otherwise, it obtains a smaller span. Thus, the adaptive nodes lead to a lower memory cost by decreasing the number of empty pointers. However, since the trie height is determined by the key length and the span size, the ART trie might be unbalanced due to the various span sizes.

For all the tries mentioned earlier, the node and span bits[6] are fixed elements. If the data set is sparsely distributed, the data structure is likely an unbalanced trie. To solve this problem, we need to explore a radix tree with nodes of equal fanout and various span bits. Meanwhile, with the increasing speed gap between cache access and main memory access, improving cache behavior becomes a crucial task to improve the performance in main memory data processing [61]. Thus, we start to pay attention to memory reference locality and cache behavior to improve the matching performance further. For a binary trie, trie search presents significant challenges due to irregular and unpredictable data accesses during trie traversal [62]. Typically the size of a block/cache line is 64 bytes [63]. The processor will read or write an entire cache line when any location in the 64-byte region is read or written [63]. Good memory reference locality leads to fewer cache misses, which can reduce the memory access time.

To leverage the memory cost and search performance, Robert Binna et al. [25] propose a height optimized trie (HOT) data structure, which retains a consistently high fanout and reduces the overall height. HOT is a new data structure designing for the main-memory database with high performance and low memory cost. A conventional trie node has a fixed span bits and data-dependent fanout, while HOT trie node features data-dependent span bits and a fixed maximum fanout[7] k [25]. The design of k is set to 32 since 32 is an optional value considering the trade-off between the cache-friendly and fast update. Thus, each trie node has the same maximum number of children, but each child may cover different bit positions.

[6]Span bits denote the specified bit positions that the trie node edge crosses.

[7]In any tree data structure, the fanout of a node is defined to be the number of children the node has. The fanout of a tree is defined to be the maximum fanout of any node in the tree.

To save memory cost, HOT stores partial key information representing this key, which introduces discriminative bits. The discriminative bit is defined as the bit position that has different values. For the integer data set, the values are "0" and "1". If any bit position has both values 0 and 1, this bit position will be included in the discriminative bits set. The partial key is extracted from the whole key using these discriminative bits. For the node layout, HOT utilizes three different sizes of the partial key: 8-bit, 16-bit, 32-bit. The data distribution determines the node type's selection since the size is equal to the number of discriminative bits among all the keys/rules in the data set. For example, comparing bit string "10" with "11", the discriminative bit is bit-0 position (least significant bit). If here comes another bit string "00", we obtain another new discriminative bit position, bit-1. If the data set only has these three rules, the 8-bit format of the node is selected since we only have two discriminative bits: bit-0 and bit-1. Of course, with the increasing size of the data set, the size of the discriminative bits set will be larger, requiring a bigger node size.

The details of insert and search operation are explained in Section 4.3. First, HOT defines the extract operation to obtain all the discriminative bits from the input data set. Next, HOT compares the extracted partial keys/rules to the inserted partial data. The comparison operations occur in both insert and search operations. HOT node has a fixed maximum of 32 children, which is efficient for utilizing SIMD instructions. It can compare one search key with 32 x 8-bit keys in parallel by utilizing AVX. HOT searches through the current data structure during the insert process to check if the rule is already inserted. If it is true, the insert operation is finished and returns 0. Otherwise, the procedure will find the missing match $BiNode$[8] and create a new discriminative bit using this missing match bit position. Therefore, each inserted rule has a unique combination of discriminative bits, demonstrating a customized span's property. HOT extracts discriminative bits from the search key during the search process and then traverses through the HOT to find if there is a matching. Since the HOT data structure is optimized for cache efficiency and allows for efficient, SIMD-optimized search [25], it is an excellent framework to reference for our wildcard matching. However, the HOT does not support arbitrary matching. This article

[8]BiNode is the binary trie node.

proposes $GenSMatcher$, which extends the HOT data structure to support arbitrary matching operations.

4.2.2 Motivation

Our focus is on wildcard matching. The goal is to find a data structure that is efficient for wildcard matching. This section discusses the need for wildcard matching and explains why we study arbitrary matching and why HOT is a good reference.

4.2.2.1 Why we study arbitrary matching?

In general, there are four different matching types: exact matching, prefix matching, range matching, and arbitrary matching. The definition of arbitrary matching is that the defined rules can be represented in any format, including the other three matching types. Therefore, arbitrary matching is a more generic matching type. From the applications aspect, in this dissertation, we mainly consider about two applications: packet classification and MPI tag matching. These applications require arbitrary matching operations since the defined rule format may have wildcard bits at any position. Furthermore, there has been little work done related to arbitrary matching. With the increasing demands of big data processing, arbitrary matching becomes more prevalent. Thus, improving the arbitrary matching performance is a crucial and challenging task.

4.2.2.2 Why we adopt HOT data structure?

For HOT data structure, their node layout is cache-friendly. Also, the insert and search operations utilize AVX instructions to take advantage of parallel computing. Most importantly, they store each rule's discriminative bits into the data structure instead of the entire rule, which reduces the overall trie height and saves memory cost. However, the HOT does not support arbitrary matching. This chapter proposes $GenSMatcher$, which extends the HOT data structure to support arbitrary matching operations.

For an arbitrary matching operation, a rule may have wildcards for an arbitrary matching operation, e.g., 101**0, where the wildcards $*$ are not located at the end of the bit string. HOT only needs to insert the discriminative bits that distinguishing between the new rule with the inserted

rules. If a rule is an arbitrary rule, the discriminative bit sets cannot be updated correctly due to the wildcard's uncertainty. To solve this problem, we propose the GenSMatcher extraction algorithm, which masks out the wildcard bits positions, excluding from its node's discriminative bits set. Therefore, each node's discriminative bits set will no longer include the wildcards bit-positions. By adding this new feature, we can deal with arbitrary rules. Since we ignore the wildcard bit-positions, we must store the wildcard rules in the leaf node to guarantee the correct matching. After these data structure modifications, the arbitrary rules can be inserted with their complete information. In addition, the search key can traverse the trie to find the accurate matching result.

4.3 GenSMatcher design

In this section, we present our GenSMatcher design for supporting arbitrary matching. We interpret rules into three fields for a wildcard rule: value, mask, and priority field defined in 1. An example of a wildcard rule set is shown in Table 4.2. In the table, we note that wildcard $*$ bit is interpreted to bit-0 in the value field and interpreted to bit-1 in the mask field. In this chapter, we define the least significant bit (LSB) position as bit-0 position. We take this wildcard ruleset as an example. The most significant bit (MSB) is bit-4. Thus, the discriminative bits set of each node can have at most five bit-positions. We will present the updated insertion and search operations in the following sections.

4.3.1 Insert operation

In order to support arbitrary matching, we design a separate procedure for insertion, which refers to *extractMask*, *extractMaskFromSuccesiveByte*, and *ExecuteForDiffingKeys* functions. Since a rule might have wildcard bits $*$, we need to ignore this wildcard bit position in a node to guarantee that all the rules' partial information inserted into the trie data structure is determinate. Therefore, we define a new function *extractMask* to extract the affected discriminative bits from the corresponding node's discriminative bits set. In this chapter, we denote the discriminative bits as *DBbits*, which are the bit positions that have a distinctive value. In the *extractMask* function,

Table 4.2: An example of a wildcard ruleset

Rules	Value	Mask	Priority
R1: 00101	00101	00000	1
R2: 10110	10110	00000	2
R3: 11110	11110	00000	3
R4: 11101	11101	00000	4
R5: 0*11*	00110	01001	5
R6: 1*10*	10100	01001	6
R7: 0*0**	00000	01011	7
R8: 0*1**	00100	01011	8
R9: 1****	10000	01111	9

first, we check if the *mask* field is 0, if it is true, return $extractMask = 0$, otherwise, we omit the wildcard bits shown in the *mask* field from the bit positions of $DBbits$.

For example, if $mask = 00101$, which shows that the bits are wildcard bits on the bit-0 and bit-2 position of a rule. If the current $DBbits = 10100$, which demonstrates that bit-2 and bit-4 positions are the distinguishing bits. Thus, we need to omit this bit-2 position since this bit position is shown in the $mask$ field. Consequently, we utilize the bitwise xor operation to omit the affected bit position. $00101 \wedge 10100 = 10001$. We note that the bit-2 of $DBbits$ becomes 0 after the xor operation, which omits the wildcard bit-2. Then, we utilize the extract instruction *_pext_u64*( ) to extract the bits position value where the bit position is "1" in $DBbits$. For doing so, we obtain the value of *extractMask*, which represents the affected bit positions excluding the wildcard bits.

Next, we process the *searchForInsert* function, which is used to check if the new rule is already inserted in the trie data structure to avoid duplicated insertion. First, we traverse the trie from the root node to the leaf node and try to find a match during this process. For doing the traversal in each level, we extract the partial rule from the inserting rule by utilizing the extract instruction *_pext_u64*$(rule.value, DBbits)$ to obtain $extractedRst$, where $DBbits$ is the discriminative bits set of the current node. If this rule's mask value is 0, then the partial rule is equal to $extractedRst$.

Otherwise, the partial rule is calculated by *_pext_u64*$(extractedRst, extractMask)$. By doing this, we omit the wildcard bits to guarantee the correct match result. If we interpreted the wildcard bit $*$ to either bit 0 or bit 1, we would lose some of the rules' information. Hence, the search result cannot always be correct. The full procedure of calculating partial rule/key of the whole rule/key is shown in Function $extractMaskFromsuccessiveByte(\ )$ in the Algorithm 2.

Algorithm 2 GenSMatcher Extraction Algorithm.

```
1: function extractMask(rule.mask, DBbits)
2:     if rule.mask == 0 then
3:         extractMask = 0
4:     else
5:         extractMask = _pext_u64((DBbits ^ rule.mask), DBbits)
6:     end if
7: end function
8:
9: function extractMaskFromsuccessiveByte(rule, DBbits)
10:    extractedRst = _pext_u64(rule.value, DBbits)
11:    if rule.mask == 0 then
12:        rst = extractedRst
13:    else
14:        rst = _pext_u64(extractedRst, extractMask)
15:    end if
16: end function
17:
```

The result of function $searchForInsert(root, rule)$ is the matched rule. Since we only insert a portion of the rule information, we need to compare the matched rule with the inserting rule by utilizing function $executeForDiffingKeys(\ )$ to obtain the new missing binary node and update the corresponding node's $DBbits$.

For the original $executeForDiffingKeys(\ )$ function, HOT compare the exact rules/keys. However, for implementing arbitrary matching during the comparisons, we need to consider the wildcard bits. Thus, the new comparison function $new\_executeForDiffingKeys(\ )$ involves two new arguments: mask field of matched rule and inserting rule, respectively, defined in Defi-

nition 5.

Definition 5. The missing match binary bit between matched rule $oldRule$ and inserting rule $newRule$ is calculated in Eq. 4.1.

$$\begin{aligned} A = &\quad oldRule.value \mid oldRule.mask \mid newRule.mask \\ B = &\quad newRule.value \mid oldRule.mask \mid newRule.mask \end{aligned}$$

$$flag\_mask = \begin{cases} 0, & \text{if } A \neq B \\ 1, & \text{if } A = B \;\&\&\; newRule.mask > 0 \end{cases} \tag{4.1}$$

where A and B are defined as the bitwise OR operation between $oldRule$ and $newRule$. The bitwise OR operation ignores all the wildcard bits from these two different rules, such that we obtain the exact discriminative bit after comparisons. The $flag\_mask$ is the identification flag for selecting different insertion procedures. If A is not equal to B and an exact discriminative bit distinguishing between $oldRule$ and $newRule$, the flag is set to be false. If A equals B, and at the same time $newRule$ has wildcard bits, this shows that a new wildcard rule needs to be inserted. In this case, the flag would be set to true. Alternatively, if the flag becomes false, we utilize HOT's insertion procedure. Otherwise, the wildcard rules will be inserted utilizing our proposed procedure. We demonstrate the details later.

First, the rule is a wildcard rule containing wildcard bits. Because we do not insert the wildcard bits position into the trie, we need to insert this wildcard rule into the affected leaf binary nodes. Therefore, we create a fixed-size array for the Binary node structure to store wildcard rules as a linked list. Second, we find the affected subtree, represented as the first index of the subtree and the subtree's number of entries. We insert the wildcard rule into these affected leaf entries.

Take the rule set in Table 4.2 for example. Here we assume $k = 32$, the LSB is the bit-0 position

and $BiNode$ is a binary node. The insertion procedure is shown in Figure 4.1. For the first rule $R1 : 00101$, we insert it into the root node in Figure 4.1 (a). For the second rule $R2 : 10110$, we find a new discriminative bit position bit-4, thus two respective $BiNode$ are created. At the same time, the bit position bit-4 is put into $DBbits$ of the root node. Also, $R1$ and $R2$ are stored into the leaf $BiNode$ entries, as shown in Figure 4.1 (b). For rule $R3 : 11110$, we extract the bits positions of $DBbits$ of the root node, that is bit-4 position of rule $R3$, and obtain the matched rule $R2$. Next, we compare the matched rule $R2$ with rule $R3$, which produces a new discriminative bit of bit-3 position. After insert the rule $R3$, the trie data structure is shown in Figure 4.1 (c). For rule $R4 : 11101$, we extract the bits of $DBbits$: bit-4 and bit-3 and obtain the matched rule $R3$. Then we compare the matched rule $R3$ with rule $R4$, which creates a new discriminative bit-1 position, as shown in Figure 4.1 (d). Note that the $DBbits$ has $\{bit\text{-}4, bit\text{-}3, bit\text{-}1\}$ so far.

For rule $R5 : 0*11*$, we extract $DBbits : \{bit\text{-}4, bit\text{-}3, bit\text{-}1\}$ and obtain $0*1$. By traversing the trie root, we obtain the matched rule $R1$. Next, we compare rule $R1$ with rule $R5$ by implementing $new\_executeForDiffingKeys(\ )$ function, and generate discriminative bit position bit-1. Because the wildcard bit $*$ of rule $R5$ is at bit-3 position, $flag\_mask$ is false such that this wildcard rule will be inserted using the original insertion procedure. For rule $R6 : 1*10*$, we extract $\{bit\text{-}4, bit\text{-}3, bit\text{-}1\}$ and obtain $1*0$, which matches with rule $R4$ that has the same bits at bit positions $\{bit\text{-}4, bit\text{-}1\}$. For comparison between rule $R4$ and rule $R6$, $oldRule$ is rule $R4$, $newRule$ is rule $R6$, and $A = 11101 \mid 00000 \mid 01001 = 11101$, $B = 10100 \mid 00000 \mid 01001 = 11101$. We note that $A = B$, thus, $flag\_mask$ is true, which triggers our new insertion procedure. First, we need to find the affected subtree, where the wildcard rule $R6$ will be inserted. From comparing rule $R4$ with rule $R6$, in the Figure 4.1 (e) we see that rule $R6$ affects the trie starting from the bit-3 position of rule $R4$'s path. Because the bit-3 on rule $R6$ is a $*$, it affects the two children "0" and "1". There is only one child on the "0" path. The leaf entry rule $R2$ is affected. There are two leaf entries on the "1" path: rule $R4$ and rule $R3$. Since the bit-1 position of rule $R6$ is 0, only rule $R4$ is affected. Therefore, the number of affected leaf entries is 2: rule $R2$ and rule $R4$.

First, we find the affected subtree, whose first entry index of the subtree is 2, rule $R2$, counting

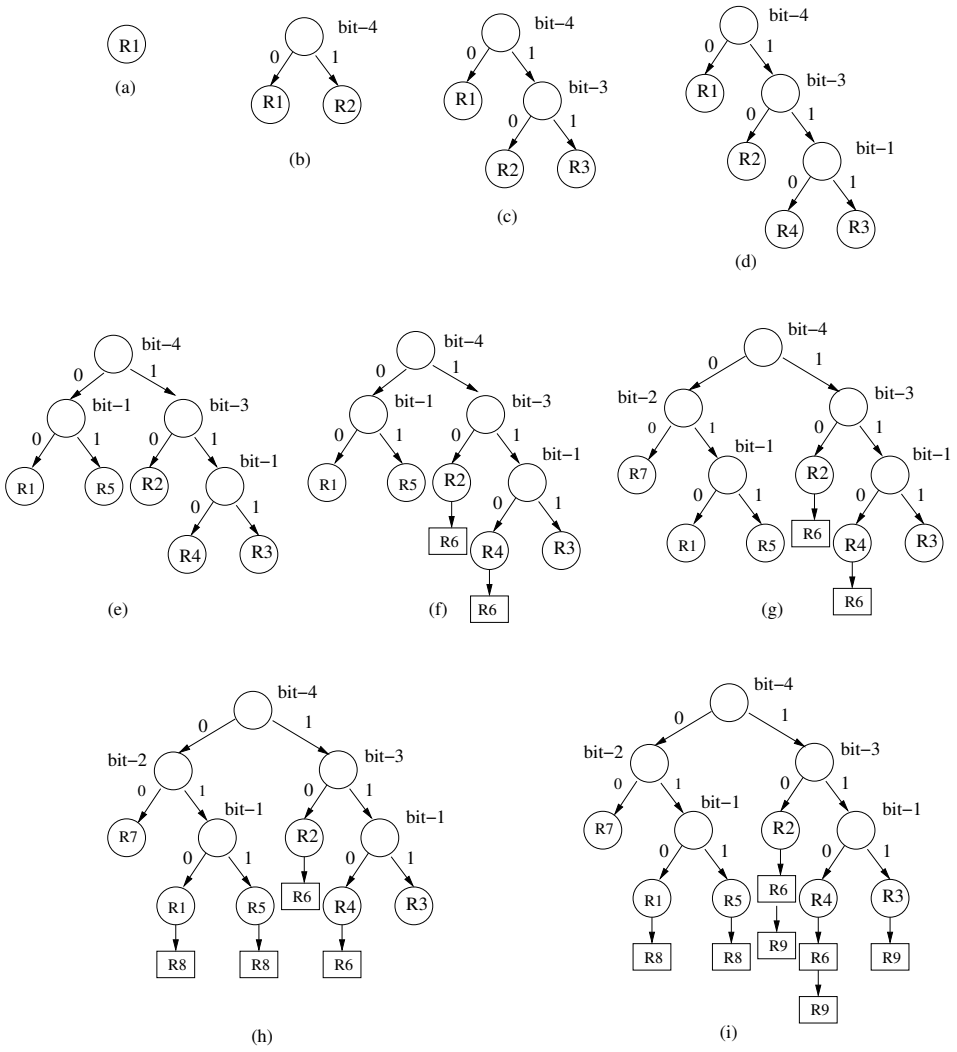


Figure 4.1: Insertion procedure of a wildcard ruleset.

from the leftmost side and index starting from 0. The number of affected entries is 2; that is, the affected leaf entries are rule $R2$ and rule $R4$. Second, we insert this wildcard rule $R6$ into the leaf entries rule $R2$ and rule $R4$. As shown in Figure 4.1 (f), the wildcard rule is represented as a rectangle.

For rule $R7 : 0 * 0 * *$, since its bit positions bit-3, bit-1 are all $*$, we only extract bit-4 and obtain matched rule $R1$. We compare rule $R1$ with $R7$ and produces a new discriminative bit position bit-2, which will be added into the root node's $DBbits$. Thus, the $DBbits$ has {bit-4, bit-3, bit-2, bit-1}. The current trie is shown in Figure 4.1 (g). For rule $R8 : 0*1**$, we extract the

Table 4.3: An example of a prefix ruleset

Rules	Value	Mask	Priority
R1: 01001	01001	00000	1
R2: 11100	11100	00000	2
R3: 11110	11110	00000	3
R4: 11011	11011	00000	4
R5: 011**	01100	00011	5
R6: 110**	11000	00011	6
R7: 00***	00000	00111	7
R8: 01***	01000	00111	8
R9: 1****	10000	01111	9

non-wildcard bit position: bit-4 and bit-2, and obtain the matched rule $R1$. We compare $R1$ with $R8$ by utilizing Eq. 4.1, $A = 00101 \mid 00000 \mid 01011 = 01111$, $B = 00100 \mid 00000 \mid 01011 = 01111$. We note that $A = B$; thus, $flag\_mask$ is true, which triggers our new insertion procedure. For rule $R8$, the affected subtree mask is 000110, which shows that the affected leaf entries are index 1 and index 2, which are rule $R1$ and rule $R5$, as shown in Figure 4.1 (g).

For the last rule $R9 : 1 * * * *$, we extract the non-wildcard bit position: bit-4, and obtain matched rule $R4$. We compare rule $R6$ with $R9$ and calculate $flag\_mask$, whose value is 1. Next, we collect the affected subtree. Since rule $R9$ has wildcards $*$ on all bits position except the bit-4, we note that the affected leaf entries are rule $R2$, rule $R4$, and rule $R3$. Thus, we insert this new wildcard rule $R9$ into the wildcard rule list under these three leaf entries. The final trie is shown in Figure 4.1 (i). Note that we insert seven wildcard rules into the wildcard rule list container of the affected leaf entries, which guarantees that all of the rules information is inserted.

For the ruleset in Table 4.2, we see that the wildcard bits $*$ are at random positions and not always at the end. Since the trie is a lexicographic data structure, different positions of wildcard bits build various data structures. This is because the rules' bits are inserted in order from the MSB to LSB. If the wildcard bit is on the upper level, this will affect more leaf entries, resulting in more insertion of wildcard rules. If the ruleset is changed to all prefix rules, the number of

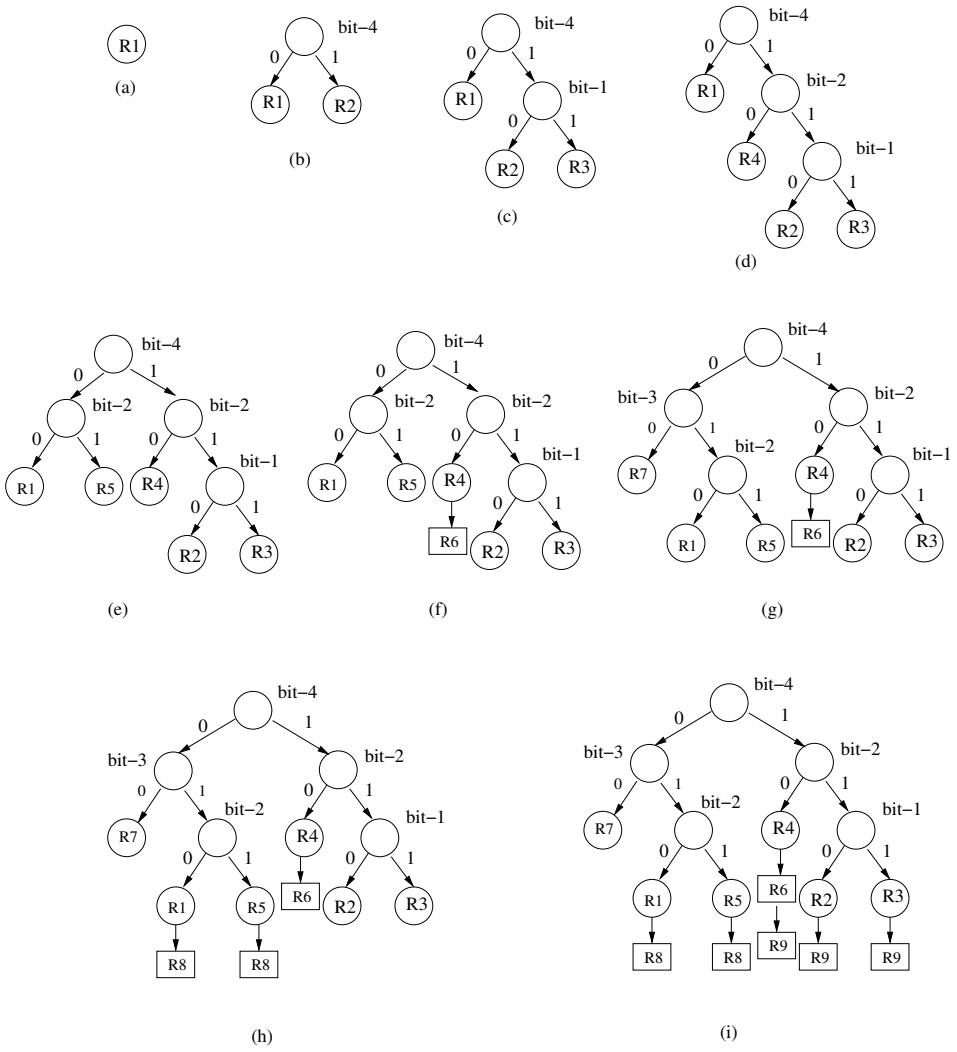


Figure 4.2: Insertion procedure of a prefix ruleset.

inserted wildcard rules will be decreased. The rule set in Table 4.3 is a transform of the ruleset in Table 4.2. Its insertion procedure is shown in Figure 4.2.

Comparing the final trie between Figure 4.2 (i) and Figure 4.1 (i), we observe that the two trie data structure are different. First, the $DBbits$ of each rule are different. We take rule $R4$ as an example. In Figure 4.1 (i) rule $R4$ is composed of $\{bit\text{-}4, bit\text{-}3, bit\text{-}1\}$, while in Figure 4.2 (i) rule $R4$ consists of $\{bit\text{-}4, bit\text{-}2\}$. Second, in Figure 4.1 (i) there are seven wildcard rules inserted into the trie, whereas in Figure 4.2 (i) only six wildcard rules are inserted. This is because Figure 4.2 (i) has all prefix rules, but Figure 4.1 (i) has arbitrary rules that have wildcard bits $*$ at any position.

However, the height of the trie is the same. Thus, in contrast to arbitrary rules, prefix rules consume less memory.

4.3.2 Search operation

For a search operation, we traverse from the root node to the leaf node of the trie. The procedure is as follows:

1. We extract the partial key from the incoming key by utilizing the search node's discriminative bits set.

2. We utilize SIMD comparison instructions to compare the extracted partial key with the trie node's entries.

3. After the comparison, if we find the matched item, we will move to the next level and go back to step (2), and we will not go to step (4) until the leaf node is reached; Otherwise, it is a non-match, return false.

4. We compare the matched entry with the incoming key in terms of the full bits to check if there is an accurate match. If it is a match, then return true. Otherwise, check if the size of the $wildcardRuleList$ is empty. If it is empty, then return false. Otherwise, the wildcard rules in the list need to be compared with the incoming key. The matching result will be the final match result.

Our proposed GenSMatcher extracts the effective discriminative bits to guarantee accurate partial rules insertion to support wildcard matching in HOT. During searching, GenSMatcher can obtain the correct match result due to the wildcard rule array.

4.4 Evaluation

This section evaluates $GenSMatcher$ on rulesets generated from packet capture (PCAP) traces and random generators. $GenSMatcher$ is our advanced arbitrary matching framework, which is SIMD and cache-friendly. We first demonstrate the evaluation methodology, followed by performance analysis. Finally, we deliver the scalability of the GenSMatcher.

4.4.1 Methodology

We integrate $GenSMatcher$ with HOT [25] data structure in C++ and employ two different rule generation methods to generate the rulesets. The rules are 64-bits integers generated from the IPv4 source and destination addresses. First, we employ a heuristic rule generation method [51] to synthesize the PCAP rulesets, where 11,000 rules are generated based on the CAIDA PCAP data [52]. We choose five various set samples with different sizes (924, 2742, 3892, 5136, 7062) and extract unique 1,044,618 keys from the traces to show the scalable performance.

Second, we use our random generator (generate random 64-bit integer) to create 2 different rule table sizes (10,000, 100,000) and 4 different key set sizes (10,000, 100,000, 100,000,000, 1000,000,000) with various match ratios: $\{0.5\%, 1\%, 2.5\%, 5\%, 10\%, 25\%, 50\%\}$, that is, the percentage of keys that find a match. With 30$\%$ wildcard rules, the number of wildcard rules is calculated as 30$\%$ $\times$ rule size. $Speedup$ is defined as GenMatcher search time / GenSMatcher search time. We evaluate our *GenSMatcher* on a real system using one single thread, in which the processor is Intel i9-9900 16-core 3.1GHz with a 32 GB DRAM. The cache hierarchy consists of a 32k L1d, 32k L1i, 256k L2, and 16384k L3 cache.

Table 4.4: Evaluation Parameters

small_keySet	10,000
big_keySet	100,000,000
bigger_keySet	1000,000,000
small_ruleSet	10,000
big_ruleSet	100,000

4.4.2 Performance Comparisons

We compare **GenSMatcher** against **GenMatcher** regarding search time, insert time, and memory cost.

4.4.2.1 *Search time:*

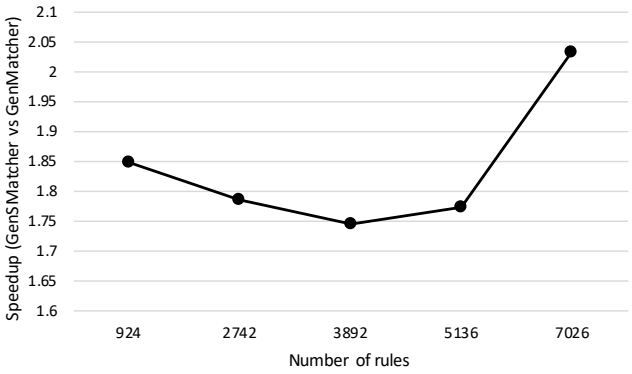


Figure 4.3: Search time performance comparison with respect to different number of rules on a small scale.

We utilize two different benchmarks to evaluate search time performance. We evaluate the performance by employing the ruleset and key set generated by PCAP traces. The search time performance is represented in Figure 4.3. This simulation evaluates five different rule sizes (924, 2742, 3892, 5136, 7062). Note that with the increasing number of rules, the speedup of **GenSMatcher** against **GenMatcher** is not linear because the trie height mainly determines search time. Table 4.5 shows the trie heights and match ratios of the corresponding test cases. With the same trie height, the larger the match ratio, the more significant the speedup. For the cases (2742, 3892, 5136, 7062), their trie height is all 3. Case 7062 has the most considerable speedup 2.03 since its match ratio has the most significant value, 37.8%. We observe that case 924 has trie height 2, which is smaller than other cases' height of 3. It was expected to outperform the other cases. However, it did not beat case 7026. This is because the match ratio of the case 924 is only 8%, which creates more comparisons.

We evaluate the search time performance by utilizing the random generated 64-bit rule benchmark on a larger scale. Figure 4.4 shows the search time speedup regarding **GenSMatcher** and **GenMatcher**. The number of rules is 10,000, the size of small_keySet is 10,000, the size

Table 4.5: Parameters of the trie

$\#Rules$	924	2742	3892	5136	7026
trie height	2	3	3	3	3
Match ratio	8%	31.1%	25.3%	29.8%	37.8%

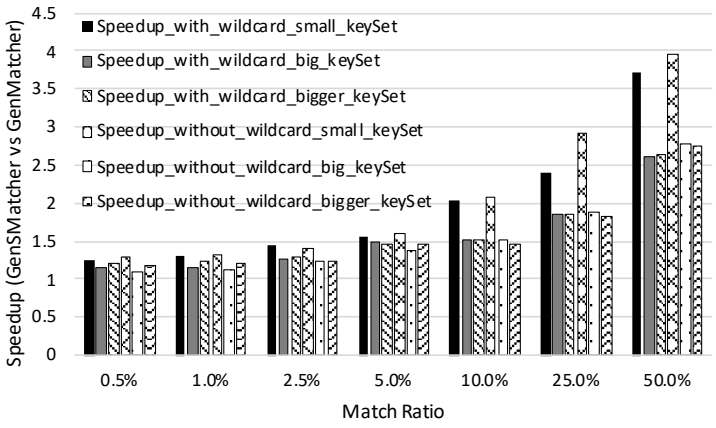


Figure 4.4: Search time performance comparison with respect to different number of keys on a large scale.

of big_keySet is 100,000,000 and the size of bigger_keyset is 1000,000,000.

First, in the figure, we see that **GenSMatcher** produces substantially better results than **GenMatcher** on both with 30% wildcard rules and without wildcard rules scenarios. This is because the search time performance of both **GenSMatcher** and **GenMatcher** are dominated by the search depth of their data structure, while the average search depth of **GenSMatcher** is smaller than **GenMatcher**.

Second, we observe that **GenSMatcher** provides more significant speedups for larger match ratios. In the GenSMatcher framework, we add a fixed-size array in the leaf node entries. This array stores the wildcard rules. During a search process, the key needs to traverse the trie from the root node to the leaf node. If the matched leaf node entry has a non-empty array, the key needs to search through the array sequentially. If the match result is a non-match, the key searches

through the whole array. Otherwise, the key might only search a portion of the array. Therefore, a non-match consumes more comparisons than a match, and thus the larger the match ratio, the greater the speedup.

Third, Figure 4.4 shows that the benchmark without wildcard rules has a slightly better performance than the benchmark with 30% wildcard rules when the match ratio is greater than or equal to 5%. This is because no array is inserted in the leaf node entries, which accelerates the search process and results in a greater speedup.

Last, we see that under a fixed-size ruleset, the speedup with small_keyset is much more significant than big_keySet and bigger_keySet. This is because of the system running overhead. The speedup is achieving stability when the key size becomes sufficiently large enough to minimize the overhead impact. Here we see no big difference between the big_keySet (100,000,000) and the bigger_keySet (1000,000,000). The search time speedup is up to 2.7X.

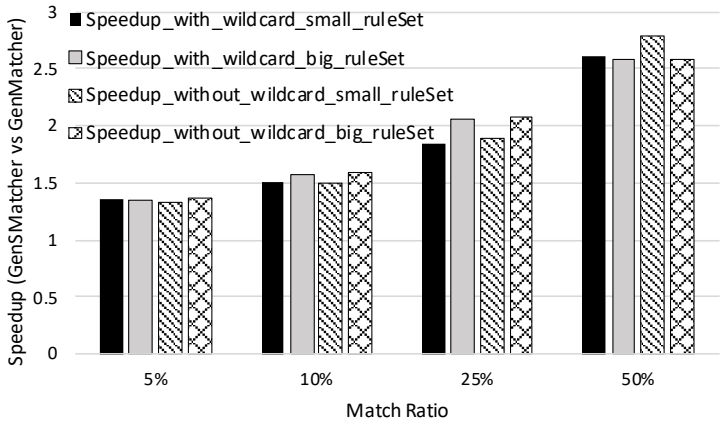


Figure 4.5: Search time performance comparison with respect to different number of rules.

Figure 4.5 shows the search time speedup concerning two different rule sizes. The number of keys is 100,000,000, the size of small_ruleSet is 10,000, and the size of big_ruleSet is 100,000. As shown in Figure 4.5, we see that the speedup performance of big_ruleSet is greater than small_ruleSet on match ratio $\{5\%, 10\%, 25\%\}$. However, on the 50% case, the big_ruleset speedup

is smaller than the small_ruleset. Since the **GenMatcher** data structure is a binary trie, each rule's depth is equal to its non-wildcard bits. The maximal depth for a rule is 64 because we use a 64-bit integer. The depth of **GenSMatcher** data structure is determined by the relationship between the rules since we build the data structure using their discriminative bits. Therefore, there is no linear trend towards the increasing number of rules, which shows the scalability of our **GenSMatcher**.

Table 4.6: The number of wildcard entries inserted in the trie

$\#Rules$	924	2742	3892	5136	7026
# Wild_rule entries	2	1235	837	1989	3783

4.4.2.2 Insert time:

We evaluate the insert time performance utilizing two sets of benchmarks. For the first set, we employ the ruleset and key set generated by PCAP traces, as shown in Figure 4.6 (a). For the other set, we utilize the ruleset and key set created by a random generator as shown in Figure 4.6 (b). Figure 4.6 (a) shows the speedup comparing **GenSMatcher** against **GenMatcher**. The insert time is determined by the number of rules and the inserted wildcard entries. In Figure 4.6 (a), note that the 924 rule case has the best speedup since the rule size is the smallest and the number of wildcard entries is only 2. Table 4.6 shows the number of wildcard entries inserted in the trie.

Figure 4.6 (b) represents the insert time performance in a larger scale. Table 4.7 shows the insert time speedup of **GenSMatcher** versus **GenMatcher** in the large scale. As shown in Table 4.7, the speedup is decreasing with the increasing number of rules. Nonetheless, inserting 5,000,000 rules is about 7.3 seconds, and the speedup is about 1.38. Thus, the insert time performance of a large-scale data set is stable.

Table 4.7: The insert time speedup between GenSMatcher and GenMatcher on a large scale

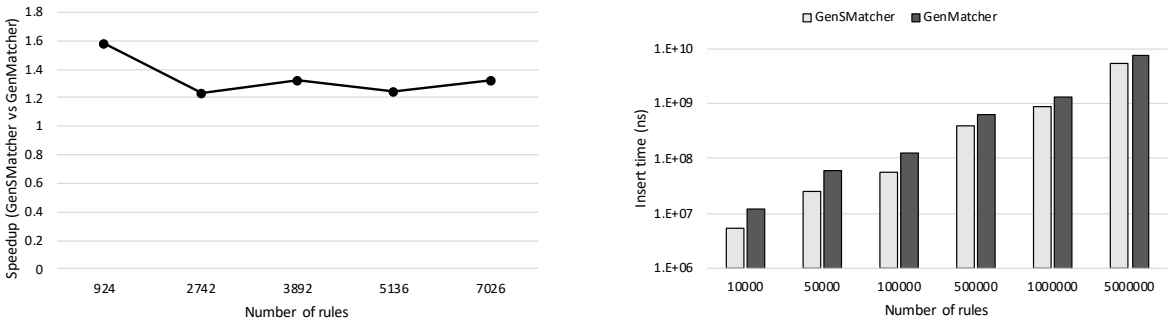
$\#Rules$	10000	50000	100000	500000	1000000	5000000
Speedup	2.31	2.37	2.14	1.56	1.48	1.38

4.4.2.3 Memory cost:

Figure 4.7 (a) shows the memory cost for 5 different rule sets on a small scale. In the figure, we note that the memory cost increases as the number of rules grows. For **GenMatcher**, we insert all the non-wildcard bits into the trie. In contrast to **GenMatcher**, **GenSMatcher** only inserts the discriminative bits into the trie. Therefore, in general, **GenSMatcher** saves more memory than **GenMatcher**. Also, **GenSMatcher** needs to insert some wildcard entries when there are no discriminative bits between the rules with wildcard bits.

Table 4.8: The memory cost comparisons between GenSMatcher and GenMatcher on a small scale

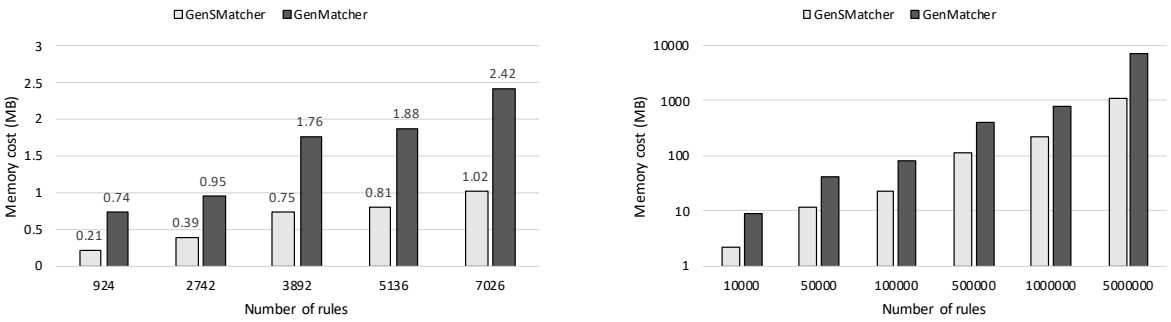
$\#Rules$	924	2742	3892	5136	7026
Reduction	3.53	2.44	2.36	2.31	2.37



(a) A small scale.

(b) A large scale.

Figure 4.6: Insert time performance comparison.



(a) A small scale.

(b) A large scale.

Figure 4.7: Memory cost comparison.

Table 4.9: The memory cost comparisons between GenSMatcher and GenMatcher on a large scale

$\#Rules$	10000	50000	100000	500000	1000000	5000000
Reduction	3.94	3.74	3.65	3.45	3.37	6.17

Figure 4.7 (b) shows the memory cost performance with respect to **GenSMatcher** and **GenMatcher** in a large scale up to 5,000,000 rules. In the figure, the memory cost is represented in log scale. From the figure we see that the trend is consistent with the Figure 4.7 (a). Table 4.8 and Table 4.9 shows the reduction of memory cost with respect to **GenSMatcher** and **GenMatcher**. Note that the reduction becomes larger with the increasing number of rules. The reduction is up to 6.17X when the number of rules is 5,000,000.

4.4.3 Scalability

In the evaluation, we utilize two different sets of benchmarks to cover different scales of rule sets and key sets. We evaluate the performance by scaling the rule size from 924 up to 5 million rules for the ruleset. For the key set, we demonstrate the performance by scaling from 10,000 up to 1,000,000,000 search operations. With respect to search time performance, compared to **GenMatcher**, **GenSMatcher** achieves up to 2.7X when the rule size is 100,000 and key size is

1000,000,000. For insert time performance, **GenSMatcher** outperforms **GenMatcher** because of the partial rules insertion and SIMD-based comparisons. The insert time is about 7.3 seconds when the rule size is 5,000,000. For memory cost, **GenSMatcher** consumes less memory than **GenMatcher**. For a rule size of 5,000,000, the memory cost is about 1.11 GB.

4.5 Conclusion

This chapter proposes $GenSMatcher$, an efficient SIMD and cache-friendly arbitrary matching mechanism. $GenSMatcher$ interprets arbitrary rules into three fields: value, mask, and priority to be able to insert into an advanced trie-based data structure. $GenSMatcher$ employs our proposed extraction algorithm to process the wildcard bits and insert arbitrary rules with randomly positioned wildcards. To guarantee accurate match results with wildcard rules, $GenSMatcher$ adds an array of wildcard entries to the leaf entries, which stores the wildcard rules. Experiments show that GenSMatcher achieves search time speedup by utilizing the SIMD feature on average by 2.7X compared to GenMatcher, and up to 6.17X reduction for the memory footprint.

5.1 Introduction

MPI is a famous parallel programming model for developing parallel scientific and big data applications [64]. Its implementations rely upon rapid sender/receiver matching to achieve high throughput messaging. With the increasing requirements of big data applications reliant on MPI, it is crucial to improve the matching throughput.

For the message tuple (c, s, t), note that the contexID/communicator c restricts the rank/process space, and the rank s restricts the tag space for a given request [65]. There are mainly three different data structures for both posted receive queue (PRQ) and unexpected message queue (UMQ):

- Linked-list-based design: There is one big linked list for both PRQ and UMQ. This data structure can easily guarantee the order semantic. Most open-source MPI libraries such as MVAPICH2, MPICH, and OpenMPI typically use a simple doubly linked list data structure to maintain the requests posted by the application [66]. The matching operation complexity of a best-case is $O(C)$, where C is a small constant. However, the search time increases as the queue length grows. It is not scalable concerning the speed of operation. The matching operation complexity is $O(N)$ on a worst-case, where N is the linked list length.

- Rank-based design: The design allocates a linked list for each rank ID. There will be n linked lists, where n is the number of processes in a job. The job's size determines memory cost. That is, the memory requirements grow linearly with the number of processes. For this method, we need to carefully trade off the need for performance with the memory overhead to achieve the best performance and scalability [66].

- Bin-based design: This design employs hash tables to allocate messages into different bins. The search complexity is $O(1)$, but collisions can occur. Compared to rank-based design, the bin-based design tries to improve the performance of search time under a limited memory capacity.

Zounmevo et al. [65] proposed a multidimensional queue traversal mechanism whose operation time and memory overhead grow sub-linearly with the job size. This data structure is designed only for large message queues. Flajslik et al. [67] proposed a bin-based data structure for tag matching. This paper utilized a hash map to reduce search time for matches in the PRQ and UMQ. To maintain the required MPI ordering semantics, they also utilized a globally ordered list to preserve the order for UMQ data structure. If the application posts many received operations with wildcard, the posted receive messages need to be searched through the linked list. Thus, this scheme is not scaled with the increasing number of wildcard receive messages. Since there will be some wildcard messages, the rank-based and bin-based design need to use a different linked list to store the wildcard messages. As the number of wildcard messages grows, the search performance will be degraded significantly.

Bayatapour et al. [66] proposed a design that allows the MPI library to adapt to different communication patterns and dynamically switch to the most appropriate design to deliver the best performance with minimal overhead. In [66], the scheme always starts with the default double linked list design. Once the average number of messages surpasses the threshold value, the design is switched to the bin-based structure or the rank-based structure. However, they did not give any detail on how to choose the threshold. Also, they did not analyze the overhead for switching among the different schemes.

In order to improve the matching performance and to guarantee the semantic requirement, this chapter proposes a hybrid data structure to reduce the impact of wildcard messages.

5.2 Motivation

Emerging HPC applications impose strict requirements for data processing delays and throughput. Message matching performance is the key to achieve high throughput. The application sends messages and receives messages to perform the tag matching operations and complete the communications between the cores in an interconnection network. Since the received message may have a wildcard on the source or tag field, there might be multiple matches between the receive messages and sending messages. To guarantee the correct communications between processes,

we need to preserve the order of semantic rules such that the set with the highest priority sending and receiving messages will always be the correct match result when there have multiple match candidates.

As the application scales up, it requires a more extensive interconnection network that includes more cores and utilizes more processes, which generates a more complex PRQ and UMQ. The challenge of MPI tag matching is to achieve high performance while guaranteeing the order semantic. Currently, linked lists are a traditional data structure to store all the messages and guarantee communication between processes across cores run successfully. However, with more extensive applications, the length of linked lists becomes very large, and the search performance is significantly degraded. Besides, there is a lack of data structures and approaches that support wildcard messages. Thus, it is crucial to develop the tag matching mechanism to process wildcard messages efficiently.

5.3 Design

In our design, we use 4-tuple (p, c, s, t) instead of (c, s, t). p represents the insertion order of the message sequence, including sender message and receiver message.

5.3.1 Hybrid Data Structure Design

The preserve order semantic operation occurs in two cases:

- For (SM, PRQ) match, when one sender message matches with multiple receiver messages in PRQ, we pick the oldest receiver message.
- For (RM, UMQ) match, when one receiver message matches multiple sender messages in UMQ, we pick the oldest sender message.

As the receiver messages are stored into PRQ, there are four types of messages in PRQ, as shown in Table 5.1. In order to improve the matching performance while preserving the semantic order, we proposed a hybrid data structure composed by a trie and a hash map. We utilize the trie data structure to take advantage of the wildcard messages. Thus, we split all 4 types of messages

into two parts: one part stored into trie, the other part stored into hash map, as shown in Table 5.2 and Table 5.3 respectively.

Table 5.1: Message types in PRQ

c	s	t
1	$*$	$*$
2	2	$*$
2	2	3
2	$*$	3

Table 5.2: Message types in PRQ_T

c	s	t
1	$*$	$*$
2	2	$*$

Table 5.3: Message types in PRQ_H

c	s	t
2	2	3
2	$*$	3

Since the sending message are stored in UMQ, there is only 1 type of message in UMQ. The fields of the tuple are all specified with no wildcard.

5.3.2 PRQ Matching Framework

As shown in Fig. 5.1, there is two various data structure for PRQ matching, which are bin-based data structure (PRQ_H) and trie-based data structure (PRQ_T). In PRQ, if the message is a MPI_ANY_TAG, it will be inserted into the PRQ_T. Otherwise, it will be inserted into the PRQ_H. Thus, $PRQ = PRQ\_H \bigcup PRQ\_T$.

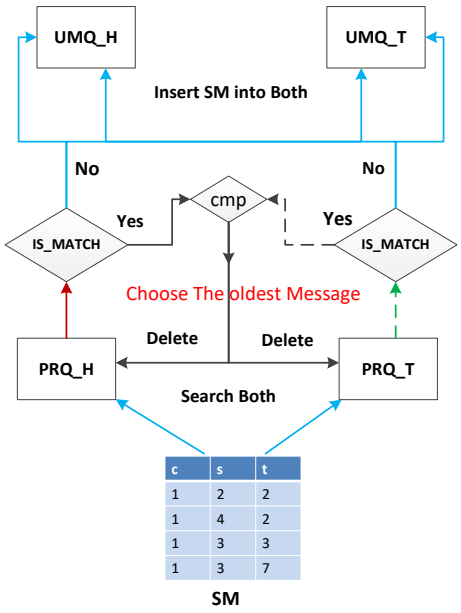


Figure 5.1: Send message side matching framework

For the bin-based data structure, we employ a hash function to allocate the messages into different bins. If a collision occurs in any bin, the message will be stored in the bin as a linked list. For the trie-based data structure, we insert the corresponding messages as a prefix format. Since the PRQ entries are inserted into two different data structures, the search operations need to be performed in PRQ_T and PRQ_H in parallel to preserve the semantic order. If the search result from PRQ_T and PRQ_H both match, we need to compare the matched sequence ID and choose the oldest message. If neither of the search results matches, we need to insert the SM into PRQ_T and PRQ_H.

5.3.3 PRQ_T Data Structure

According to our MPI application, there are only two types of message in PRQ_T, as shown in Table 5.2. Note that, the two types are all prefix format: $(c, s, *)$ and $(c, *, *)$. The search complexity is $O(C)$, which C is 2 or 3. From the message field view, we need to search the field c and s sequentially. The trie has three levels, in which the first level includes all the c values. All the s values are covered in the second level. The third level represents the sequence order for each message. $O(log_q^{(l_c+l_s)})$. The total memory cost is calculated as:

$$MEM = trieNode.size * trieNode.count \tag{5.1}$$

where

$$trieNode.size = keyArray.size + pointerArray.size$$

If there are lots of null pointers in the pointer array, this will result in a big memory waste.

Our goal is to decrease memory waste while guaranteeing search performance. We adopt the adaptive node size idea from the ART paper [26]. The node size is determined by the number of children they have. Our adaptive node trie data structure has three levels and two types of trie nodes: inner node and leaf node. On the first level, the nodes can only be an inner nodes. On the second level, the nodes can be either inner node or leaf node. On the third level, the nodes can only be a leaf node, which stores the message's priority. Since the length of a message can only be two values, we do not need to make a balanced tree. Instead, we need to save memory costs to make the data structure space more efficient.

Note that the first level represents the data distribution of c. The number of communicators determines the size of the root node's children. The second level represents the data distribution of s. The size of the node is determined by the number of processes in each corresponded communicator.

For all the messages in PRQ_T, we need to do some pre-operations before building the trie data structure. Since the radix tree is an index structure, we need to sort the node by their integer value. First, we sort the c value in ascending order in PRQ_T. After sorting, we can get the number of communicators, which determines the number of root node's children. Second, for each different c value, we sort the s value, which determines the node size. Since all the messages will be inserted in order, we do not need to sort the third level values. In the end, we obtain ordered messages in PRQ_T.

We apply six different trie node types for our trie data structure: Node4, Node16, Node32, Node64, Node128, and Node256. The pointer size is 8 bytes. The key size is 1 byte.

- Node4: can store up to 4 child pointers. The node consists of an array of key-value and an array of pointers for children. The size of each array is 4.
- Node16: can store up to 16 child pointers. The node includes an array of key values and an array of pointers of size 16. A key can be searched in parallel utilizing SIMD instructions.
- Node32: can store up to 32 child pointers. The node includes an array of key-value and an array of pointers with a size 32. A key can be found with parallel comparisons using SVE-256 bit instructions.
- Node64: can store up to 64 child pointers. The size of the key array and pointer array is 64. A key can be searched in parallel comparisons using SVE-512 bit instructions.
- Node128: can store up to 128 child pointers. The size of the key array and pointer array is 128. A key can be searched in parallel comparisons using SVE-1024 bit instructions.
- Node256: can store up to 256 child pointers. The size of the key array and pointer array is 256. A key can be searched in parallel comparisons using SVE-1024 bit instructions.

Table 5.4 shows the above six node types' attributes, where n is the number of keys in the node.

Table 5.4: Node types in trie data structure

Type	Children	Memory (bytes)	Search mode	Search complexity
Node4	2-4	4+4·8	Serial	O(n)
Node16	5-16	16+16·8	SVE-128 bit	O(1)
Node32	17-32	32+32·8	SVE-256 bit	O(1)
Node64	33-64	64+64·8	SVE-512 bit	O(1)
Node128	65-128	128+128·8	SVE-1024 bit	O(1)
Node256	129-256	256+256·8	SVE-1024 bit	O(1)

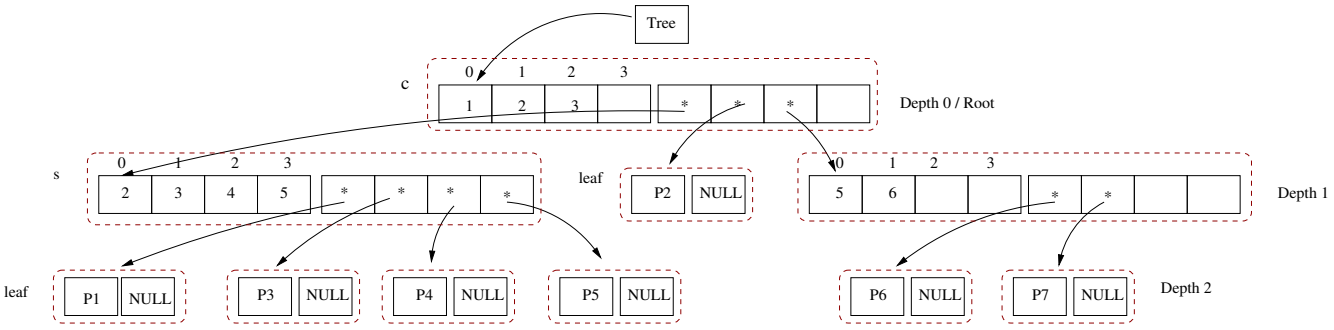


Figure 5.2: The detail data structure for PRQ_T

We choose the suitable node size to build the trie to minimize the total memory cost. After we sort the message in PRQ_T, we can know how many children each node has. For the field c, the node type is determined by the number of communicators. For the field s, the node type is determined by the number of different processes.

For the leaf node, we store the priority/sequence number into the leaf. Every leaf node has a unique value to preserve the semantic order. The data distribution in PRQ_T determines which node type and how many node types we need to choose to minimize the total memory cost.

Take Table 5.5 as an example. Three different fields represent the message: p, c, s, representing message sequence ID, communicator ID, and process rank ID, respectively.

The detailed data structure is shown in Fig. 5.2. For this PRQ_T example, we utilize Node4[1]

[1]Note that the root has three children, communicator 1 has four children, and communicator 3 has two children.

Table 5.5: PRQ_T entries

p	c	s
P1	1	2
P2	2	
P3	1	3
P4	1	4
P5	1	5
P6	3	5
P7	3	6

with a key array and a pointer array[2]. For all the messages in PRQ_T trie data structure, there are two different lengths of the message: 8-bit and 24-bit. In Fig. 5.2, the message P2's length is 8, which has the c part. Other messages are all 24-bit, which are all leaf nodes at the last level.

In this chapter, we determine the inner node type according to the number of children of their parents, minimizing the memory waste. For the leaf node, we store the sequence ID of the message into the leaf node.

5.3.4 PRQ_H Data Structure

There are two types of messages in PRQ_H, as shown in Table 5.3. Note that one type is filled with all specified integers, and one type has a wildcard in the source field. These two types of messages are inserted into a hashmap data structure. Since there may be a message with a wildcard in the source field, the hash function is given as:

$$hash(c, s, t) = (c + t) \,\%\, NUM\_BINS \tag{5.2}$$

Thus, communicator 1 and communicator 3 are Node4 type. They have a key array and a pointer array with a size of 4.

[2]The pointer in a pointer array is represented as $*$ in Fig. 5.2

5.3.5 UMQ Matching Framework

UMQ stores the unexpected messages, including all defined messages, including filed c, s, and t. If we want to use trie data structure in UMQ, we need to insert all the messages into both UMQ_T and UMQ_H data structure, as shown in Fig. 5.3. When the application posts a new receive message, the search operation is performed through UMQ_T if the received message is a MPI_ANY_SOURCE. Otherwise, the search operation is performed through UMQ_H. For the insertion of UMQ, the UMQ_H and UMQ_T are built in parallel. The search operation is just performed in one data structure, which depends on the receiver message type.

The UMQ_T data structure is similar to the PRQ_T. Since the messages in UMQ are all specified tuples, the only difference is the length of messages. The length of messages in UMQ_T has only one value, which equals the sum of all the length of c, s, and t. Thus, the leaf node of UMQ_T can just appear at the last level. The UMQ_H data structure is the same as the PRQ_H. Note that the search operation happens between RM and UMQ_H. Since RM might have a wildcard, UMQ_H utilizes the same hash function as the PRQ_H.

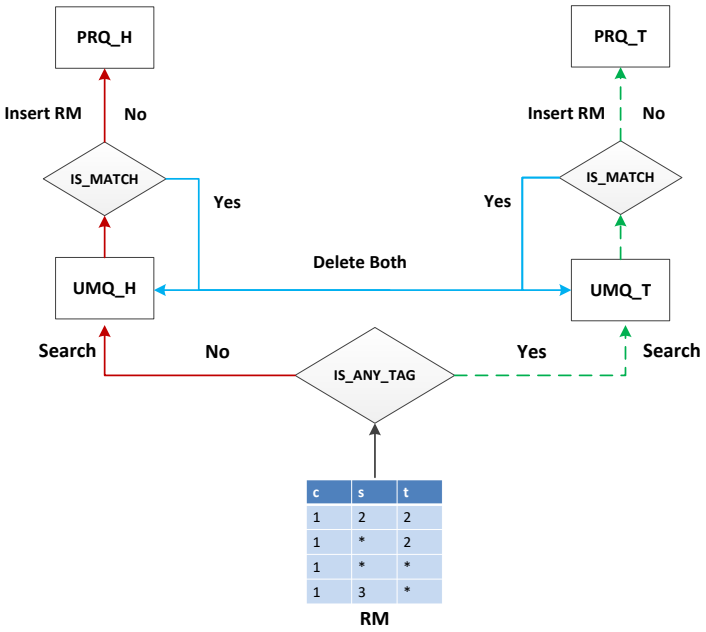


Figure 5.3: Receiver message side matching framework

If the search results is a match, we need to delete the matched entry in both UMQ_H and UMQ_T, otherwise, we need to insert RM into the corresponded PRQ. If the RM has a wildcard in the tag field, we insert RM into PRQ_T. Otherwise, we insert RM into PRQ_H.

5.4 Evaluation

In this section, we evaluate the proposed hybrid data structure on our generated micro-benchmark, and NAS Parallel Benchmark (NPB) [68] applications and PICSARlite [69] application. We first present the evaluation methodology. Secondly, we compare the results against the previous approaches.

5.4.1 Methodolody

We program the MPI tag matching framework in C and generate our micro-benchmark in Python. In our micro-benchmark, we randomly generate received messages with $anySource$ and $anyTag$ field. Messages used in this project consist of tuples (c, s, t). Each field is represented as an integer. Besides, we record the orders for each message in order to guarantee the semantic order.

We generate our application for the micro-benchmark consisting of two lists of messages: Sending Message (SM) and Receive Message (RM). To learn more about the impact of different application traces, we generate different maximum depths of the queue for an application (32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072). We evaluated our hybrid data structure on a system using one single thread, where the processor is Intel Xeon E5-2697A V4 32-core 2.6GHz with a 512 GB DRAM. The cache hierarchy consists of a 32k L1d, 32k L1i, 256k L2, and 40960k L3 cache. We evaluate the performance in search time, the number of search attempts in PRQ and UMQ, and memory cost. We compare the hybrid data structure against three different data structures for MPI tag matching:

- Baseline: Linked-list data structure.
- MPI_list: Array + Linked-list data structure.

- Intel: Hashmap data structure.

For the NPB application and PICSARlite application, we integrate our proposed Hybrid method and Intel method into the MPICH-3.2.1 library. The MPICH library utilizes the Linked-list data structure to implement the tag matching process, which is represented as **Lib** in the performance figures. We implement **Hybrid** and **Intel** method in the MPICH library and compare the search time performance. We evaluate Integer Sort (IS) benchmarks in NAS parallel benchmark 3.3.1. In our evaluation, we run the class C problem size. The evaluations are conducted on the Ada High-Performance research Computing cluster at Texas A&M University. The Ada cluster has 793 general compute nodes equipped with Intel Xeon E5-2670 v2 (Ivy Bridge-EP), 10-core, 2.5GHz processors. Nodes are connected through FDR-10 Infiniband host channel adapters.

5.4.2 Microbenchmark performance

We compare the **Hybrid** data structure against the previous three different data structures regarding search time, search attempt, and memory cost.

5.4.2.1 Search time:

We run scaling simulations using the generated point-to-point workload with up to 131072 messages in a queue. The scaling results of execution time are presented in Figure 5.4, and Figure 5.5 shows the speedup of our proposed hybrid matching algorithm over the **Baseline**, **Intel**, and **MPI_list** for varying maximum depth of the queue. Figure 5.4 shows that our **Hybrid** performance is getting better with the increasing maximum depth of the queue. Note that the **Hybrid** outperforms the best when the depth is reaching 4096. Compared to the **Baseline**, the speedup is up to 50X. Compared to the **MPI_list**, the speedup is up to 2X. We see that our proposed **Hybrid** outperforms the best with the increasing depth of the queue. For all matching algorithms, the search time is dominated by the number of search attempts. As we can see in Figure 5.6, our proposed **Hybrid** match algorithm has the best performance indicated by its small number of search attempts.

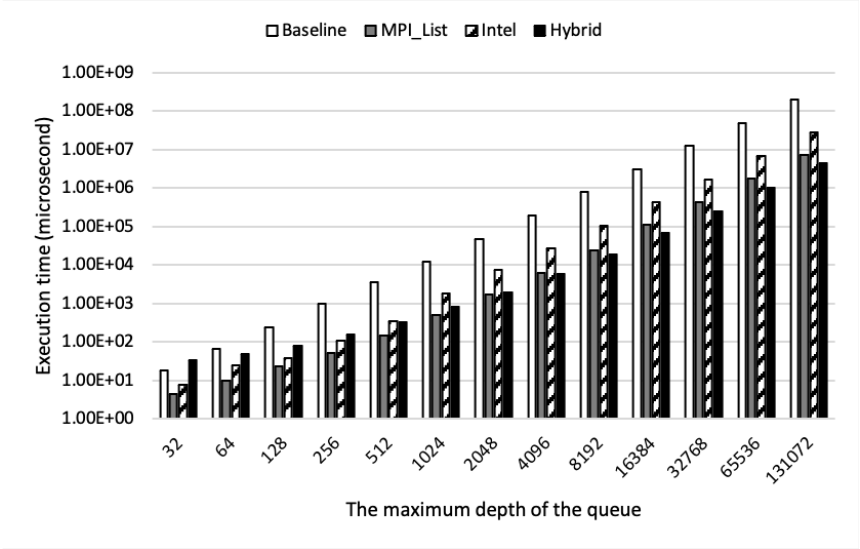


Figure 5.4: Execution time.

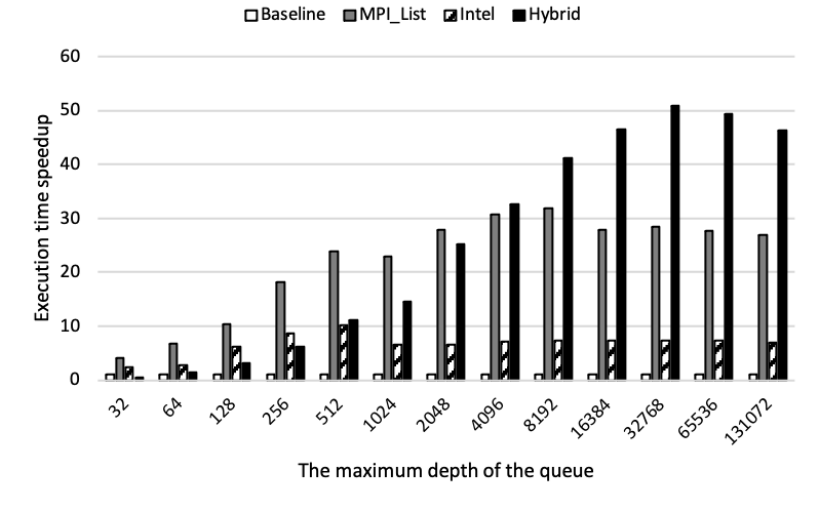


Figure 5.5: Search time speedup.

5.4.2.2 Search attempt:

Figure 5.6 shows the total search attempts for point-to-point communications. We evaluate the search attempts as the number of comparisons. The **Baseline** utilizes a single linked list. Note that the **Baseline** has the most significant number of search attempts. With the the increasing

maximum depth of the queue, the length of the linked list grows. Accordingly, the number of search attempts is increasing with the increased length of the linked list. Since our proposed **Hybrid** matching algorithm utilizes tree search, the search attempt is much smaller than traversing an extensive linked list with a larger queue, which results in the best performance. This result is consistent with the search time performance.

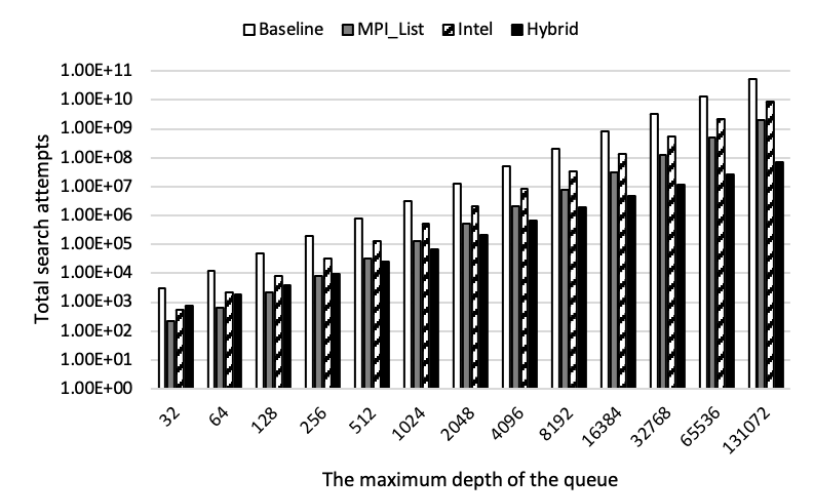


Figure 5.6: Total search attempt.

5.4.2.3 Memory cost:

Figure 5.7 represents the memory cost for the four different matching algorithms. Note that, the memory cost of **Baseline** and **MPI_list** are the same. This is because both of them have the same total length of items. Since our generated benchmark has almost 30% of wildcard messages on the receiver side, the **Intel** matching algorithm has two large linked-lists on both PRQ and UMQ. Thus, the memory cost of **Intel** has the worst performance. For our proposed **Hybrid** matching algorithm, the memory cost is close to the **Baseline** and **MPI_list** when the queue depth is small. As the process count increases, the memory cost of the **Hybrid** exceeds the **Baseline** and **MPI_list** method. Since the goal of our proposed **Hybrid** matching algorithm

is to improve the search performance under a reasonable memory cost, the memory cost is the tradeoff.

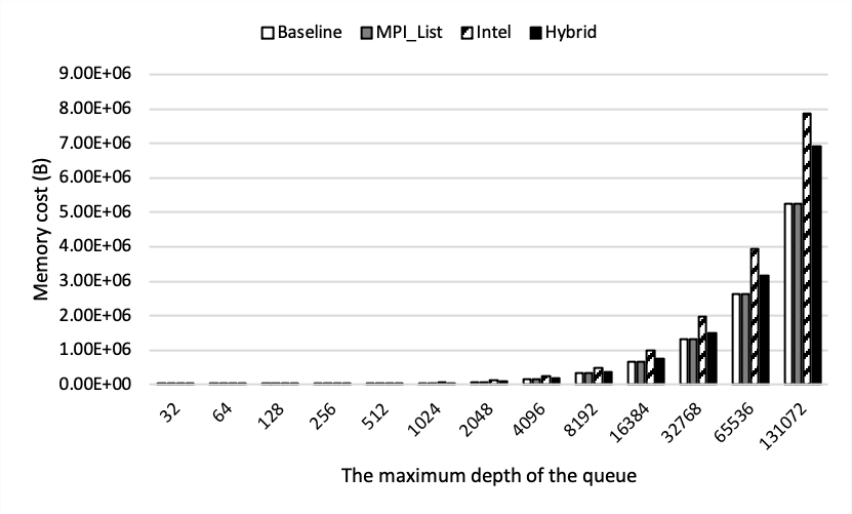
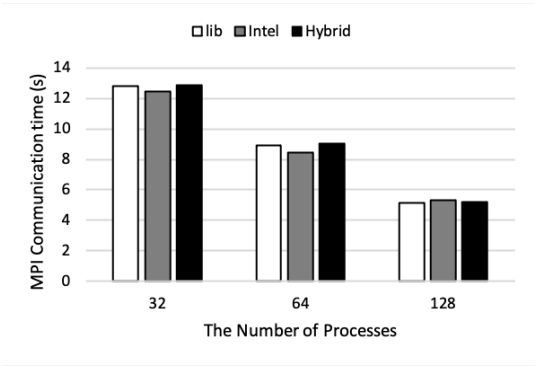


Figure 5.7: Memory cost.

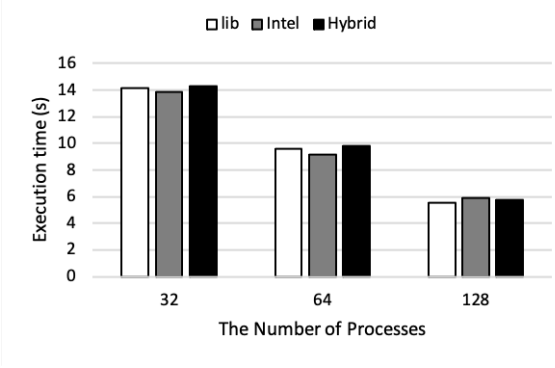
5.4.3 NPB benchmark performance

This section evaluates our proposed **Hybrid** mechanism by utilizing existing NPB application benchmarks. We use **Lib** as the baseline and depict the time speedup of **Hybrid** and **Intel** versus **Lib**. Figure 5.8 shows the time performance of the IS benchmark with respect to **Hybrid**, **Lib**, and **Intel** at 3 different numbers of processes (32, 64, 128). The execution time is the total time, including the communication time and computation time. The communication time is the sum of time spent in MPI for communication and synchronization, which occurred in the MPI library. The computation time is the sum of time spent outside the MPI library. Figure 5.8 (a) shows the MPI communication time performance comparisons and Figure 5.8 (b) represents the execution time performance comparisons.

The IS benchmark performs all-to-all communication and leverages MPI_Alltoall collective operations. Therefore, Figure 5.9 shows that the number of search attempts of PRQ and UMQ increases with a more significant number of processes. In Figure 5.8, we see that the execution time



(a) MPI communication time.



(b) Execution time.

Figure 5.8: NPB Benchmark IS performance comparisons with respect to various number of processes.

and MPI communication time are decreasing with the increasing number of processes, which is because there are more processes involved in parallel processing the application, which reduces the computation time. For the MPI communication time, from Figure 5.10 we see that the queue length of **Lib** is not increasing linearly as the process count grows. Therefore, the MPI communication time is not increasing with a larger size of processes. For the **Intel** method, we see that its PRQ length is 0 since there have no wildcard messages in IS benchmark.

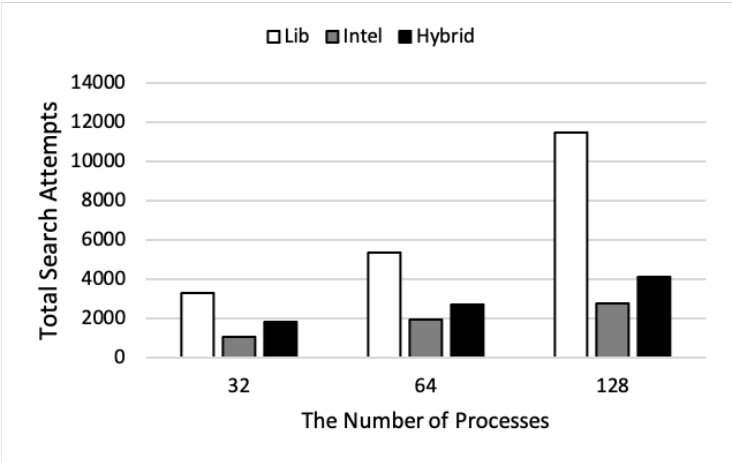


Figure 5.9: Total search attempts over IS Benchmark.

	Lib_PRQ_L	Lib_UMQ_L	Intel_PRQ_L	Intel_UMQ_L
32	32	31	0	113
64	32	32	0	143
128	32	66	0	140

Figure 5.10: PRQ and UMQ queue length over IS Benchmark.

5.4.4 PICSARlite benchmark performance

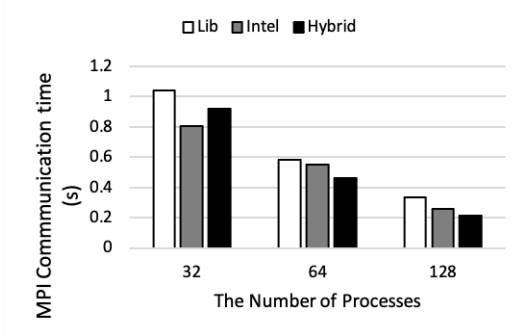
PICSARlite [69] is a subset of the PICSAR suite that allows testing smaller electromagnetic Particle-In-Cell kernels. This application makes use of point-to-point communications, which transmits messages between a pair of processes where sender and receiver cooperate with each other [70], that is, two-sided communication. To the best of my knowledge, there are not many applications involving wildcards in receive operations. PICSARlite is one of these applications using MPI_ANY_TAG wildcard messages.

	Lib_PRQ_L	Lib_UMQ_L	Intel_PRQ_L	Intel_UMQ_L
32	54	44	9	48
64	60	35	9	60
128	118	98	9	71

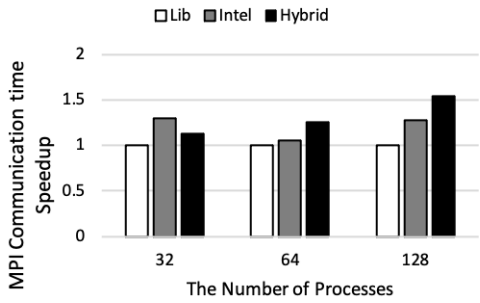
Figure 5.11: PRQ and UMQ queue length over PICSARlite Benchmark.

Figure 5.12 represents the PICSARlite performance concerning three different matching mechanisms over three different counts of processes (32, 64, 128). **Lib** is the baseline. Figure 5.12 (a) shows the MPI communication time comparisons, where we see that the MPI communication time is reducing with the increasing number of processes due to the increasing queue depth of the PRQ and UMQ of **Lib** and **Intel** method, as shown in Figure 5.11. In the figure, we see that the PRQ and UMQ of **Lib** are increasing with a large number of processes. For the **Intel**, the depth of PRQ is not changing, but the UMQ depth is increasing. However, for our proposed **Hybrid**, the tree depth is not increasing linearly with the increasing number of processes. Thus, Figure 5.12

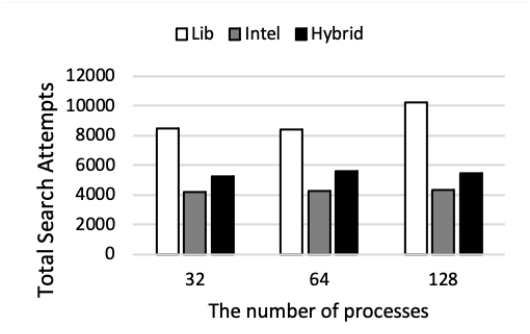
(b) shows the MPI communication time speedup performance, where the **Hybrid** is increasing with a larger number of processes. From the figure, we see that in contrast to the **Lib**, **Hybrid** has the best speedup performance on cases 64 and 128. In the figure, we see that **Hybrid**'s speedup is up to 1.55X. Figure 5.12 (c) shows the total search attempts performance. From the figure, you can see that **Lib** increases significantly on 128 processes while **Hybrid** does not change much due to their different data structures. Figure 5.12 (d) represents the total execution time performance, including the MPI communication time and the computation time. In the figure we note that **Hybrid** outperforms **Lib** and **Intel**. **Lib** has the worst performance since it utilizes the linked list for its tag matching data structure.



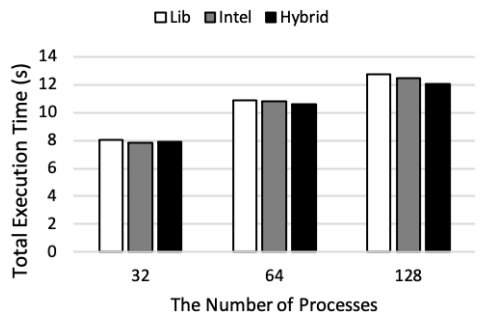
(a) MPI communication time.



(b) MPI communication time speedup.



(c) Total search attempts.



(d) Execution time.

Figure 5.12: PICSARlite Benchmark performance comparisons with respect to various number of processes.

5.5 Conclusions

This chapter proposes a new hybrid data structure and matching mechanism to reduce matching operation time, processed in PRQ and UMQ. The hybrid data structure is composed of a trie and hash map. We evaluate our mechanism on our generated micro-benchmark and existing MPI applications over varying numbers of processes. We compare our proposed mechanism with the baseline and Intel scheme on a single node and an HPC cluster, respectively. Experiment results show that our proposed Hybrid outperforms the baseline and intel mechanism in terms of MPI communication time. For the PICSARlite application, the MPI communication time speedup is up to 1.55 X.

6. CONCLUSION

High-performance computing usage has been increasing due to the high demand for big data applications, not only for scientific applications but also in commodity applications. In an HPC network, the HPC cluster is a system of multiple interconnected nodes in a switched network [58]. At each node, the MPI is the communication protocol of parallel processes in HPC networks which pass messages over the network to synchronize and coordinate each process's results. Also, the node is connected through an SDN-based switch, in which the forwarding hardware is decoupled from the control decision, control plane. SDN-based HPC networks can program the control plane for different application requirements and choose the optimal resources and configuration to satisfy users' various performance requirements. Since high-performance matching is a vital part of improving the performance of HPC networks, it is essential to study the acceleration techniques for high-performance matching.

This dissertation discussed the arbitrary matching problems on packet classification and MPI tag matching applications. First, we propose GenMatcher: a generic software-only arbitrary matching mechanism for fast and efficient searches under a limited memory threshold [3]. Since GenMatcher employs our proposed mapping and grouping approaches to assign the arbitrary rules with the most significant similarities into the same group, it generates a minimal number of groups within a memory threshold [3]. It can build a binary trie to perform fast binary searches. Second, we introduce GenSMatcher: an efficient SIMD and cache-friendly arbitrary matching mechanism. GenSMatcher takes advantage of the SIMD instruction and modern cache features that exploit data localities to accelerate the searches and reduce memory usage. Third, we present the hybrid matching framework: integrating a trie and hash map data structure for MPI tag matching. Our hybrid mechanism can process wildcard messages more efficiently.

The proposed high-performance arbitrary matching mechanisms can be deployed in other matching applications, such as string data. With the growth of enormous data application demands and the diverse QoS requirements, these generic arbitrary matching mechanisms become

fundamental techniques for high-performance matching.

REFERENCES

[1] S. M. Ghazimirsaeed, R. E. Grant, and A. Afsahi, "A dedicated message matching mechanism for collective communications," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[2] T. Shen, D. Zhang, G. Xie, and X. Zhang, "Optimizing multi-dimensional packet classification for multi-core systems," *J. Comput. Sci. Technol.*, vol. 33, no. 5, pp. 1056–1071, 2018.

[3] P. Wang, L. McHale, P. V. Gratz, and A. Sprintson, "Genmatcher: A generic clustering-based arbitrary matching framework," *ACM Trans. Archit. Code Optim.*, vol. 15, Nov. 2018.

[4] T. Inoue, T. Mano, K. Mizutani, S. I. Minato, and O. Akashi, "Rethinking packet classification for global network view of software-defined networking," in *2014 IEEE 22nd International Conference on Network Protocols*, pp. 296–307, Oct 2014.

[5] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Transactions on Networking*, vol. 24, pp. 1251–1264, April 2016.

[6] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich, "Efficient FIB representations on distributed platforms," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–14, 2017.

[7] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid CPU-FPGA architectures," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), pp. 403–415, ACM, 2017.

[8] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich, "FIB efficiency in distributed platforms," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pp. 1–10, Nov 2016.

[9] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, pp. 8–23, Mar 2001.

[10] C. R. Meiners, A. X. Liu, and E. Torng, "Bit Weaving: A non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 20, pp. 488–500, April 2012.

[11] C. E. Andrade, M. G. Resende, H. J. Karloff, and F. K. Miyazawa, "Evolutionary algorithms for overlapping correlation clustering," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, (New York, NY, USA), pp. 405–412, ACM, 2014.

[12] X. Pan, D. S. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan, "Parallel correlation clustering on big graphs," *CoRR*, vol. abs/1507.05086, 2015.

[13] K. Wagstaff and C. Cardie, "Clustering with instance-level constraints," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, (San Francisco, CA, USA), pp. 1103–1110, Morgan Kaufmann Publishers Inc., 2000.

[14] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl, "Constrained k-means clustering with background knowledge," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, (San Francisco, CA, USA), pp. 577–584, Morgan Kaufmann Publishers Inc., 2001.

[15] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy sequential maximal independent set and matching are parallel on average," *CoRR*, vol. abs/1202.3205, 2012.

[16] A. Banerjee, C. Krumpelman, J. Ghosh, S. Basu, and R. J. Mooney, "Model-based overlapping clustering," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, (New York, NY, USA), pp. 532–537, ACM, 2005.

[17] C. Luo, W. Pang, and Z. Wang, *Semi-supervised Clustering on Heterogeneous Information Networks*, pp. 548–559. Cham: Springer International Publishing, 2014.

[18] Y. Gu and C. Wang, "A study of hierarchical correlation clustering for scientific volume data," in *Proceedings of the 6th International Conference on Advances in Visual Computing - Volume Part III*, ISVC'10, (Berlin, Heidelberg), pp. 437–446, Springer-Verlag, 2010.

[19] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition," in *Proc. BMVC*, pp. 81.1–81.10, 2006. doi:10.5244/C.20.81.

[20] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, (New York, NY, USA), pp. 169–178, ACM, 2000.

[21] M. P. Forum, "MPI: A message-passing interface standard," tech. rep., University of Tennessee, USA, 2012.

[22] N. Askitis and R. Sinha, "HAT-trie: A cache-conscious trie-based data structure for strings," in *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, (AUS), p. 97–105, Australian Computer Society, Inc., 2007.

[23] M. Mäsker, T. Süß, L. Nagel, L. Zeng, and A. Brinkmann, "Hyperion: Building the largest in-memory search tree," in *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, (New York, NY, USA), p. 1207–1222, Association for Computing Machinery, 2019.

[24] D. R. Morrison, "PATRICIA-practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, p. 514–534, Oct. 1968.

[25] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A height optimized trie index for main-memory database systems," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), p. 521–534, Association for Computing Machinery, 2018.

[26] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 38–49, April 2013.

[27] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: Fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), p. 339–350, Association for Computing Machinery, 2010.

[28] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pp. 174–183, Oct 2004.

[29] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification," *SIGCOMM Comput. Commun. Rev.*, vol. 42, p. 335–346, Aug. 2012.

[30] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP lookup performance with FIB explosion," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 39–50, Aug. 2014.

[31] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pp. 111–122, 2013.

[32] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10, Sept 2016.

[33] P. Lemarinier, K. Hasanov, S. Venugopal, and K. Katrinis, "Architecting malleable MPI applications for priority-driven adaptive scheduling," in *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, (New York, NY, USA), pp. 74–81, ACM, 2016.

[34] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, April 2008.

[35] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 18, pp. 490–500, April 2010.

[36] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "Split: Optimizing space, power, and throughput for TCAM-based classification," in *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pp. 200–210, Oct 2011.

[37] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *2014 IEEE 22nd International Conference on Network Protocols*, pp. 308–319, Oct 2014.

[38] C. L. Hsieh and N. Weng, "Many-field packet classification for software-defined networking switches," in *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 13–24, March 2016.

[39] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, (New York, NY, USA), pp. 15–26, ACM, 2014.

[40] Y. Qu, S. Zhou, and V. K. Prasanna, "Scalable many-field packet classification on multi-core processors," in *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pp. 33–40, Oct 2013.

[41] Y. Qi, B. Xu, F. He, X. Zhou, J. Yu, and J. Li, "Towards optimized packet classification algorithms for multi-core network processors," in *2007 International Conference on Parallel Processing (ICPP 2007)*, Sept 2007.

[42] H. Lu and S. Sahni, "$o(logw)$ multidimensional packet classification," *IEEE/ACM Transactions on Networking*, vol. 15, pp. 462–472, April 2007.

[43] S. I. Nikolenko, K. Kogan, G. Rétvári, E. R. Bérczi-Kovács, and A. Shalimov, "How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 521–526, April 2016.

[44] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.1," 06 2015. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[45] Y. K. Sia, H. G. Goh, S. Y. Liew, and M. L. Gan, "Spanning multi-tree algorithm for node and traffic balancing in multi-sink wireless sensor networks," in *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pp. 2190–2195, Aug 2015.

[46] D. Drachsler, M. Vechev, and E. Yahav, "Practical concurrent binary search trees via logical ordering," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, (New York, NY, USA), pp. 343–356, ACM, 2014.

[47] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, (New York, NY, USA), pp. 161–171, ACM, 2012.

[48] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Building scalable virtual routers with trie braiding," in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, March 2010.

[49] H. Lim and H. Y. Byun, "Packet classification using a bloom filter in a leaf-pushing area-based quad-trie," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, (Washington, DC, USA), pp. 183–184, IEEE Computer Society, 2015.

[50] J. Lee, H. Byun, J. H. Mun, and H. Lim, "Utilizing 2-D leaf-pushing for packet classification," *Computer Communications*, vol. 103, pp. 116 – 129, 2017.

[51] L. Mchale, J. Case, P. V. Gratz, and A. Sprintson, "Stochastic pre-classification for SDN data plane matching," in *Proceedings of the 2014 IEEE 22Nd International Conference on Network Protocols*, ICNP '14, (Washington, DC, USA), pp. 596–602, IEEE Computer Society, 2014.

[52] "The CAIDA Anonymized 2012 Internet Traces - 2012, Kc Claffy, Dan A ndersen, Paul Hick." http://www.caida.org/data/passive/passive_2012_dataset.xml.

[53] A. Fiessler, S. Hager, and B. Scheuermann, "Flexible line speed network packet classification using hybrid on-chip matching circuits," in *2017 IEEE 18th International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–8, June 2017.

[54] H. Alimohammadi and M. Ahmadi, "Common non-wildcard portion-based partitioning approach to sdn many-field packet classification," *Computer Networks*, vol. 181, p. 107534, 2020.

[55] O. N. Foundation, "Openflow switch specification," tech. rep., ONF, USA, 2015.

[56] S. Shirali-Shahreza and Y. Ganjali, "Rewiflow: Restricted wildcard openflow rules," *SIGCOMM Comput. Commun. Rev.*, vol. 45, p. 29–35, Sept. 2015.

[57] T. Shen, D. Zhang, G. Xie, and X. Zhang, "Optimizing multi-dimensional packet classification for multi-core systems," *J. Comput. Sci. Technol.*, vol. 33, no. 5, pp. 1056–1071, 2018.

[58] S. V. Krishna, A. Shrivastava, and S. J. Wagh, "SDN in high performance computing for scientific and business environment (SBE)," in *2017 International Conference on Computational Intelligence in Data Science(ICCIDS)*, pp. 1–8, 2017.

[59] S. Date, H. Abe, D. Khureltulga, K. Takahashi, Y. Kido, Y. Watashiba, P. U-Chupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, "An empirical study of sdn-accelerated hpc infrastructure for scientific research," in *Proceedings of the 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, ICCCRI '15, (USA), p. 89–96, IEEE Computer Society, 2015.

[60] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, "KISS-tree: Smart latch-free in-memory indexing on modern architectures," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, (New York, NY, USA), p. 16–23, Association for Computing Machinery, 2012.

[61] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), p. 78–89, Morgan Kaufmann Publishers Inc., 1999.

[62] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, (USA), p. 285, IEEE Computer Society, 1999.

[63] "Why software developers should care about CPU caches." https://medium.com/software-design/why-software-developers-should-care-about-cpu-caches-8da04355bb8a.xml.

[64] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for mpi over infiniband: Design alternatives and benefits," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, (New York, NY, USA), p. 32–39, Association for Computing Machinery, 2006.

[65] J. A. Zounmevo and A. Afsahi, "An efficient MPI message queue mechanism for large-scale jobs," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pp. 464–471, Dec 2012.

[66] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10, Sept 2016.

[67] M. Flajslik, J. Dinan, and K. D. Underwood, *Mitigating MPI Message Matching Misery*, pp. 281–299. Cham: Springer International Publishing, 2016.

[68] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 158–165, ACM, 1991.

[69] "Picsarlite benchmark." https://proxyapps.exascaleproject.org/app/picsarlite.xml.

[70] N. Sultana, M. Rüfenacht, A. Skjellum, P. Bangalore, I. Laguna, and K. Mohror, "Understanding the use of message passing interface in exascale proxy applications," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e5901, 2020.