Multi-core Implementation of Decomposition-based Packet Classification Algorithms[1]

Shijie Zhou, Yun R. Qu, and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering, University of Southern California
Los Angeles, CA 90089, U.S.A.
{shijiezh, yunqu, prasanna}@usc.edu

Abstract. Multi-field packet classification is a network kernel function where packets are classified based on a set of predefined rules. Many algorithms and hardware architectures have been proposed to accelerate packet classification. Among them, decomposition-based classification approaches are of major interest to the research community because of the parallel search in each packet header field. This paper presents four decomposition-based approaches on multi-core processors. We search in parallel for all the fields using linear search or range-tree search; we store the partial results in a linked list or a bit vector. The partial results are merged to produce the final packet header match. We evaluate the performance with respect to latency and throughput varying the rule set size (1K ~ 64K) and the number of threads per core (1 ~ 12). Experimental results show that our approaches can achieve 128 ns processing latency per packet and 11.5 Gbps overall throughput on state-of-the-art 16-core platforms.

1 Introduction

Internet routers perform *packet classification* on incoming packets for various network services such as network security and Quality of Service (QoS) routing. All the incoming packets need to be examined against predefined rules in the router; packets are filtered out for security reasons or forwarded to specific ports during this process. Moreover, the emerging Software Defined Networking (SDN) requires *OpenFlow table lookup* [1], [2], which is similar as the classic multi-field packet classification mechanism. All these requirements make packet classification a kernel function for Internet.

Many hardware and software-based approaches have been proposed to enhance the performance of packet classification. One of the most popular methods for packet classification is to use Ternary Content Addressable Memories (TCAMs) [3]. TCAMs are not scalable and require a lot of power [4]. Recent work has explored the use of Field-Programmable Gate Arrays (FPGAs) [5]. These designs can achieve very high throughput for moderate-size rule set, but they also suffer long processing latency when external memory has to be used for large rule sets.

Use of software accelerators and virtual machines for classification is a new trend [6]. However, both the growing size of the rule set and the increasing bandwidth of the Internet make memory access a critical bottleneck for high-performance packet classification. State-of-the-art multi-core optimized microarchitectures [7], [8] deliver a number of new and innovative features that can improve memory access performance. The increasing gap between processor speed and memory speed was bridged

[1] Supported by U.S. National Science Foundation under grant CCF-1116781.

by caches and instruction level parallelism (ILP) techniques [9]. For cache hits, latencies scale with reductions in cycle time. A cache hit typically introduces a latency of two or three clock cycles, even when the processor frequency increases. The cache misses are overlapped with other misses as well as useful computation using ILP. These features make multi-core processors an attractive platform for low-latency network applications. Efficient parallel algorithms are also needed on multi-core processors to improve the performance of network applications.

In this paper, we focus on improving the performance of packet classification with respect to throughput and latency on multi-core processors. Specifically, we use decomposition-based approaches on state-of-the-art multi-core processors and conduct a thorough comparison for various implementations. The rest of the paper is organized as follows. Section 2 formally describes the background and related work, and Section 3 covers the details of the four decomposition-based algorithms. Section 4 summarizes performance results, and Section 5 concludes the paper.

2 Background

2.1 Multi-field Packet Classification

Table 1. Example rule set

ID	SA	DA	SP	DP	PROT	PRI	ACT
1	175.77.88.155/32	119.106.158.230/32	0 - 65535	6888 - 6888	0x06	1	Act 0
2	175.77.88.6/32	36.174.239.222/32	0 - 65535	1704 - 1704	0x06	2	Act 1
3	175.77.88.4/32	36.174.239.222/32	0 - 65535	1708 - 1708	0x06	3	Act 0
4	95.105.143.51/32	39.240.26.229/32	0 - 65535	1521 - 1521	0x06	4	Act 2
5	95.105.143.51/32	204.13.218.182/32	0 - 65535	0 - 65535	0x01	5	Act 3
6	152.175.65.32/28	248.116.141.0/28	24032 - 24032	123 - 123	0x11	5	Act 5
7	17.21.12.0/23	224.0.0.0/5	0 - 65535	0 - 65535	0x00	6	Act 4
8	233.117.49.48/28	233.117.49.32/28	750 - 750	123 - 123	0x11	7	Act 3

Multi-field packet classification problem [10] requires five fields to be examined against the rules: 32-bit source IP address (SIP), 32-bit destination IP address (DIP), 16-bit source port number (SP), 16-bit destination port number (DP), and 8-bit transport layer protocol (PROT). In SIP and DIP fields, longest prefix match is performed on the packet header. In SP and DP fields, the matching condition of a rule is a range match. The PROT field only requires exact value to be matched. We denote this problem as the classic packet classification problem in this paper. A new version of packet classification is the OpenFlow packet classification [2] where a larger number of fields are to be examined. In this paper we focus on the classic packet classifi-

cation, although our solution techniques can also be extended to the OpenFlow packet classification.

Each packet classification engine maintains a rule set. In this rule set, each rule has a rule ID, the matching criteria for each field, an associated action (ACT), and/or priority (PRI). For an incoming packet, a match is reported if all the five fields match a particular rule in the rule set. Once the matching rule is found for the incoming packet, the action associated with that rule is performed on the packet. We show an example rule set consisting of 8 rules in Table 1; the typical rule set size (denoted as *N*) ranges from 50 to 1K [10].

A packet can match multiple rules. If only the highest priority one needs to be reported, it is called *best-match* packet classification [11]; if all the matching rules need to be reported, it is a *multi-match* packet classification. Our implementation is able to output both results.

2.2 Related Work

Most of packet classification algorithms on general purpose processors fall into two categories: *decision-tree based* and *decomposition based* algorithms.

The most well-known decision-tree based algorithms are HiCuts [12] and HyperCuts [13] algorithms. The idea of decision-tree based algorithms is that each rule defines a sub-region in the multi-dimensional space and a packet header is viewed as a point in that space. The sub-region which the packet header belongs to is located by cutting the space into smaller sub-regions recursively. However, searching in a large tree is hard to parallelize and it requires too many memory accesses. Therefore it is still challenging to achieve high performance using efficient search tree structure on state-of-the-art multi-core processors.

Decomposition based algorithms usually contain two steps. The first step is to *search* each field individually against the rule set. The second step is to *merge* the partial results from all the fields. The key challenge of these algorithms is to parallelize individual search processes and handle merge process efficiently. For example, one of the decomposition based approaches is the Bit Vector (BV) approach [17]. The BV approach is a specific technique in which the lookup on each field returns an *N*-bit vector. Each bit in the bit vector corresponds to a rule. A bit is set to "1" only if the input matches the corresponding rule in this field. A bit-wise logical AND operation gathers the matches from all fields in parallel.

The BV-based approaches can achieve 100 Gbps throughput on FPGA [19], but the rule set size they support is typically small (less than 10K rules). Also, for port number fields (SP and DP), since BV-based approaches usually require rules to be represented in ternary strings, they suffers from range expansion when converting ranges into prefixes [20].

Some recent work has proposed to use multi-core processors for packet classification. For example, the implementation using HyperSplit [14], a decision-tree based algorithm, achieves a throughput of more than 6Gbps on the Octeon 3860 multi-core platform. However, the decomposition based approaches on state-of-the-art multi-core processors have not been well studied and evaluated.

3 Algorithms

We denote the rules in the classification rule set as *original rules*. Given a 5-field rule set, we present the procedure of our decomposition-based approaches in 3 phases:

- Preprocess: The prefixes in IP address fields or exact values in PROT field are translated into ranges first; then all the rules are projected onto each field to produce a set of non-overlapping subranges in each field. We denote the rule specified by a subrange as a *unique rule*. Rules having the same value in a specific field are mapped into the same unique rule in this field. For example, the 8 rules in Table 1 can be projected into 5 unique rules in the SP field: [0, 749], [750, 750], [751, 24031], [24032, 24032], and [24033, 65535]. We either construct *linked-lists* using the unique rules for classification, or build a *range-tree* using the unique rules. We show an example for preprocessing 4 rules in a specific field in Figure 1.
- Search: The incoming packet header is also split into 5 fields, where each of the header field is searched individually and independently. We employ *linear search* or *range-tree search* for each individual field, and we record the partial matching results of each field by a *rule ID set* or a *Bit Vector* (BV).
- Merge: The partial results from different fields are merged into the final result. For partial results represented by a rule ID set, merge operation is performed using linear merge; for partial results represented by a BV, merge operation is performed using a bitwise logical AND.

Depending on the types of search phase and the representation of partial results, we have various implementations. Section 3.1 and 3.2 introduce the algorithms for linear search and range-tree search, respectively, while Section 3.3 and 3.4 cover the set representation and BV representation of partial results. Section 3.5 summarizes all the implementation.

3.1 Linear Search

A linear search can be performed in each field against unique rules. We denote the match of each unique rule as a *unique match*. We consider linear search due to the following reasons:

- Linear search can be used as a baseline to compare various decomposition-based implementations on multi-core processors.
- The number of unique rules in each field is usually less than the total number of the original rules [15].
- For multi-match packet classification problem (see Section 2), it requires *O(N)* memory to store a rule set consisting of *N* rules, while it also requires *O(N)* time to obtain all the matching results in the worst case. Linear search is still an attractive algorithm from this viewpoint.
- Linear search results in regular memory access patterns; this in turn can lead to good cache performance.

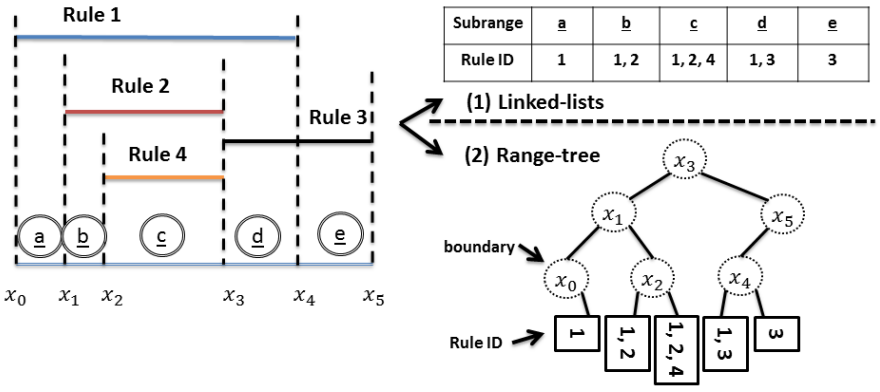


Figure 1. Preprocessing the rule set to (1) linked-lists or (2) a balanced binary range-tree

To optimize the search, we construct linked-lists for linear search. Linked-list is a data structure which specifies a unique rule and its associated original rules. In Figure 1, we show an example of constructing linked-lists from original rules in a specific field. As shown in Figure 1, all the 4 rules are preprocessed into 5 linked-lists due to the overlap between different original rules; during the search phase, the incoming packet is examined using the unique rules (comparing the input value sequentially with the subrange boundaries $x_0, x_1, \ldots, x_5$); the partial results are then stored for merge phase using either a rule ID set (Section 3.3), or a BV (Section 3.4).

3.2 Range-tree Search

The key idea of range-tree search is to construct a balanced range tree [16] for efficient tree-search; the range-tree search can be applied to all the fields in parallel.

To construct a range-tree, we first translate prefixes or exact values in different packet header fields into ranges. Overlapping ranges are then flattened into a series of non-overlapping subranges as shown in Figure 1. A balanced binary search tree (range-tree) is constructed using the boundaries of those subranges.

For example, in Figure 1, the value in a specific field of the input packet header is compared with the root node (boundary x_3) of the range tree first. Without loss of generality, we use inclusive lower bound and exclusive upper bound for each subrange. Hence there can be only two outcomes from the comparison: (1) the input value is less than x_3, or (2) the input value is greater than or equal to x_3. In the first case, the input value is then compared with the left child (x_1) of the root node; in the second case, the input value is compared with the right child (x_5) of the root node. This algorithm is applied iteratively until the input value is located in a leaf node representing a subrange. Then the partial results can be extracted for future use in the merge phase.

This preprocess explicitly gives a group of non-overlapping subranges; the input can be located in a subrange through the binary search process. However, as shown in Figure 1, each subrange may still correspond to one or more valid matches.

3.3 Set Representation

After the search in each individual field is performed, we can represent each partial result using a set. As shown in Figure 2, for the input located in the subrange c, a set consisting of 3 rule IDs {1, 2, 4} is used to record the partial matching result in a field. We denote such a set as a *rule ID set*. We maintain the rule IDs in a rule ID set in sorted order. Thus, for the partial result represented using a rule ID set, all the rule IDs are arranged in ascending (or descending) order.

As shown in Algorithm 1, the merging of those partial results is realized by linear merge efficiently: we iteratively eliminate the smallest rule ID value of all 5 fields unless the smallest value appears in all the 5 fields. A common rule ID detected during this merge process indicates a match. The correctness of Algorithm 1 can be easily proved; this algorithm is similar as the merge operation for two sorted lists in merge sort, except 5 rule ID sets need to be merged.

We show an example of merging 5 rule ID sets in Figure 2. Initially we have 5 rule ID sets as the partial results from 5 fields. In the first iteration, since the rule IDs are stored in each rule ID set in ascending order, the first rule IDs in 5 rule ID sets (smallest rule IDs, namely, Rule 1 in SIP field, Rule 2 in DIP field, Rule 1 in SP field, Rule 3 in DP field, and Rule 3 in PROT field) are compared; since they are not the same, we delete the minimum ones from the rule ID sets: Rule 1 is deleted from the SIP rule ID set, and Rule 1 is deleted from the SP rule ID set. We iteratively apply this algorithm to delete minimum rule IDs in each iteration, unless all the minimum rule IDs of 5 fields are the same; a common rule ID appearing in all 5 fields (such as Rule 3 in Figure 2) indicates a match between the packet header and the corresponding rule. We record the common rule IDs as the final result.

```
Algorithm 1: (Set S_0) MERGE (Set S_1, Set S_2, ..., Set S_5)
if any of the input sets is null then
return (null)
else
 for non-null input Set S_i do in parallel
   a_i → first element of S_i
   end for
   while (a_i is not the last element of S_i) do
if not all a_i's are equal then
  S_i → delete the minimum a_i from S_i
  MERGE (Set S_1, Set S_2, ..., Set S_5)
else
  push a_i into set S_0
  S_i → delete a_i from S_i for all i
  MERGE (Set S_1, Set S_2, ..., Set S_5)
      end if
   end while
   return (Set S_0)
  end if
```

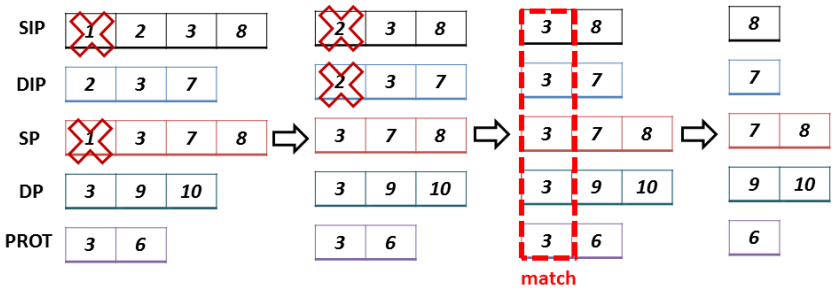


Figure 2. Merging 5 rule ID sets

3.4 BV Representation

Another representation for the partial results is the BV representation [17]. For each unique rule/subrange, a BV is maintained to store the partial results. For a rule set consisting of N rules, the length of the BV is N for each unique rule/subrange, with each bit corresponding to the matched rules set to 1. For example, in Figure 1, the matched rule IDs for the subrange c can be represented by a BV "1101", indicating a match for this subrange corresponds to Rule 1, Rule 2 and Rule 4.

Both linear search and range-tree search generate 5 BVs as partial results for 5 fields, respectively. The partial results can be merged by bitwise AND operation to produce the final result. In the final result, every bit with value 1 indicates a match in all the five fields; we report either all of their corresponding rules (multi-match) or only the rule with the highest priority (best-match).

3.5 Implementations

In summary, we have four implementations: (1) Linear search in each field, with partial results represented as rule ID Sets (LS), (2) Range-tree search with partial results recorded in rule ID Sets (RS), (3) Linear search with BV representation (LBV) and (4) Range-tree search with BV representation (RBV).

We denote the total number of original rules as N, the number of unique rules as N_1 and the maximum number of original rules associated with a unique rule as N_2. Table 2 summarizes the computation time complexity for the four approaches.

Table 2. Summary of various approaches

Approach	Search	Merge
LS	$O(N_1)$	$O(N_2 \log(N))$
RS	$O(\log(N_1))$	$O(N_2 \log(N))$
LBV	$O(N_1)$	$O(N_2)$
RBV	$O(\log(N_1))$	$O(N_2)$

4 Performance Evaluation and Summary of Results

4.1 Experimental Setup

Since our approaches are not platform-dependent, we conducted the experiments on a 2× AMD Opteron 6278 processor and a 2× Intel Xeon E5-2470 processor. The AMD processor has 16 cores, each running at 2.4GHz. Each core is integrated with a 16KB L1 data cache, 64KB L1 instruction cache and a 2MB L2 cache. A 60MB L3 cache is shared among all 16 cores. The processor has access to 128GB DDR3 main memory through an integrated memory controller running at 2GHz. The Intel processor has 16 cores, each running at 2.3GHz. Each core has a 32KB L1 data cache, 32KB L1 instruction cache and a 256KB L2 cache. All 16 cores share a 20MB L3 cache.

We implemented all the 4 approaches using Pthreads on openSUSE 12.2. We analyzed the impact of data movement in our approaches. We also used *perf*, a performance analysis tool in Linux, to monitor the hardware and software events such as the number of executed instructions, the number of cache misses and the number of context switches.

We generated synthetic rule sets using the same methodology as in [18]. We varied the rule set size from 1K to 64K to study the scalability of our approaches. We used processing *latency* and overall *throughput* as the main performance metrics. We define overall throughput as the aggregated throughput in Gbps of all parallel packet classifiers. We define the processing latency as the average processing time elapsed during packet classification for a single packet. We also examined the relation between the number of threads per core and context switch frequency to study their impact on the overall performance.

4.2 Data Movement

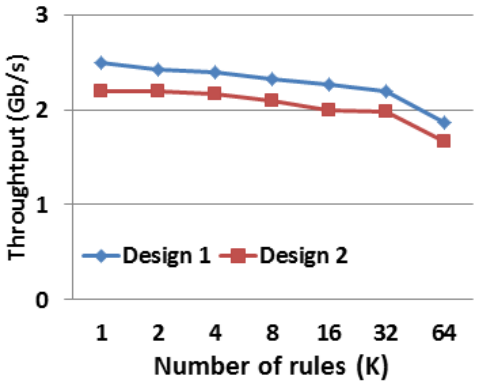
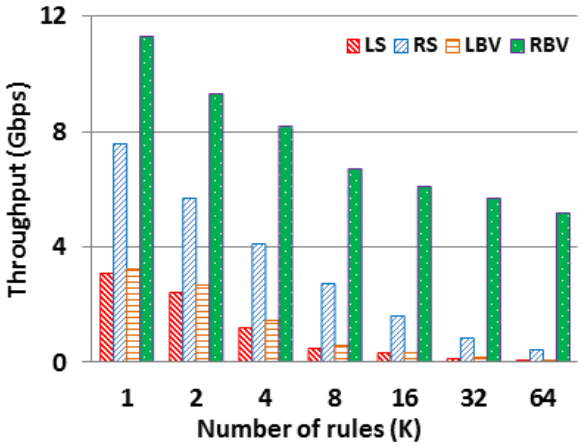


Figure 3. Data movement (Design 1: 5 cores, 5 packets; Design 2: 5 cores, 1 packet)

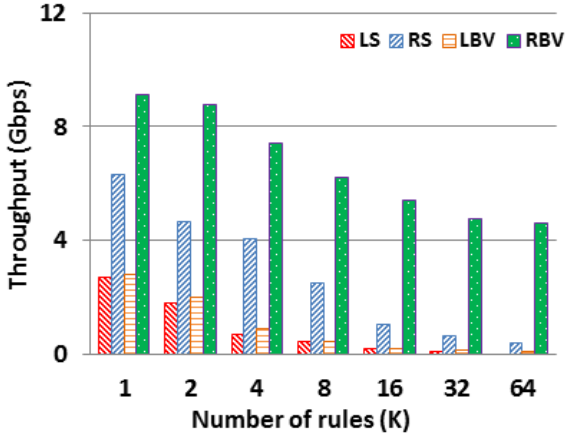
To optimize our implementation, we first conduct two groups of RBV experiments with 1K rule set: (Design 1) 5 cores process 5 packets in parallel, each processing a single packet independently; (Design 2) all the 5 cores process a single packet, each processing one packet header field. In Design 1, partial results from the five fields of

a packet stay in the same core and the search phase requires 5 range trees to be used in the same core. In Design 2, all the threads in a single core perform search operation in the same packet header field; the merge phase requires access to partial results from different cores. Our results in Figure 3 show that Design 1 has in average 10% more throughput than Design 2. In Design 2, a large amount of data move between cores, making it less efficient than Design 1. We also have similar observations in LS, RS, and LBV approaches. Therefore, we use Design 1 (a packet only stays in one core) for all of the 4 approaches, where partial results are accessed locally.

4.3 Throughput and Latency

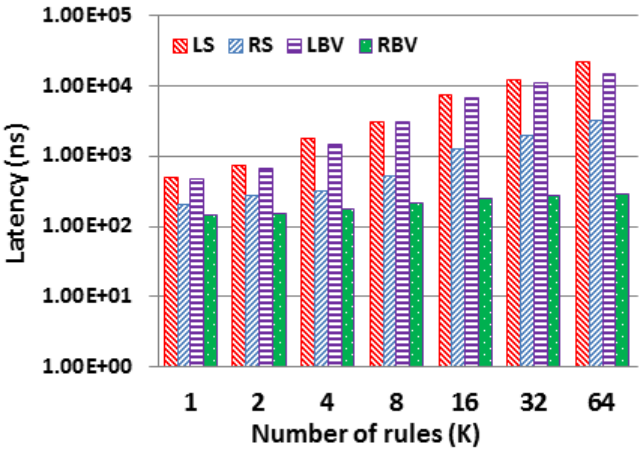


(a) Throughput on the AMD processor

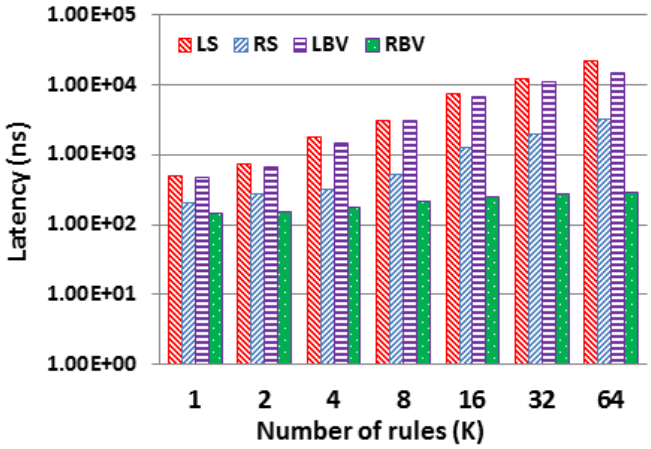


(b) Throughput on the Intel processor

Figure 4. Throughput with respect to the number of rules (*N*)



(a) Latency on the AMD processor



(b) Latency on the Intel processor

Figure 5. Latency with respect to the number of rules (*N*)

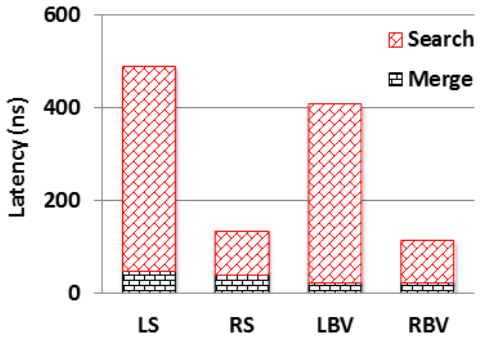
Figure 4 and Figure 5 show the throughput and latency performance with respect to various sizes of the rule set for all the four approaches. For each approach, the experiments are conducted on both the AMD and the Intel platforms. We have the following observations:

- The performance degrades as the number of rules increases. This is because when the rule set becomes larger, the number of unique rules also increases; this leads to an increase of both the search time and the merge time.
- Range-tree based approaches suffer less performance degradation when the number of rules increases. Note the time complexity for range-tree search is

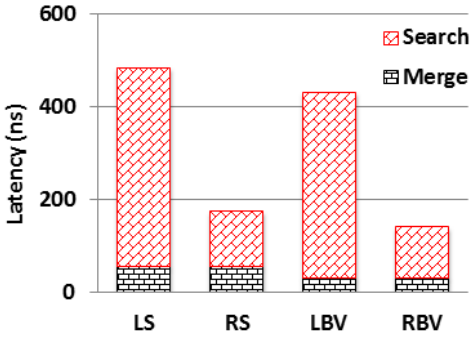
$O(\log N_1)$, while the time complexity for linear search is $O(N_1)$. For a balanced binary range-tree, when the rule set size doubles, one more tree level has to be searched, while linear search requires double the amount of search time. We show the breakdown of the overall execution time in terms of search and merge times in Section 4.4.

- Using set representation in merge phase (LS and RS) is not as efficient as using BV representation (LBV and RBV). Merging multiple rule ID sets requires $O(N_2 \log(N))$ time, while it takes $O(N_2)$ time to merge bit vectors using bitwise AND operations.

4.4 Search Latency and Merge Latency



(a) Latency on the AMD processor



(b) Latency on the Intel processor

Figure 6. Breakdown of the latency per packet on (a) the AMD multi-core processor and (b) the Intel multi-core processors (1K rule set, 5 threads per core)

We show the latency of the search and merge phases in Figure 6. We have the following observations:

- Search time contributes more to the total classification latency than the merge time, since search operations are more complex compared with the merge operations.
- As shown in Figure 5 and Figure 6, we achieve similar performance on the AMD and the Intel multi-core platforms. The performance of all the implementations on Intel machine is slightly worse than the performance on the AMD machine. Note that the Intel machine has a lower clock rate.

4.5 Cache Performance

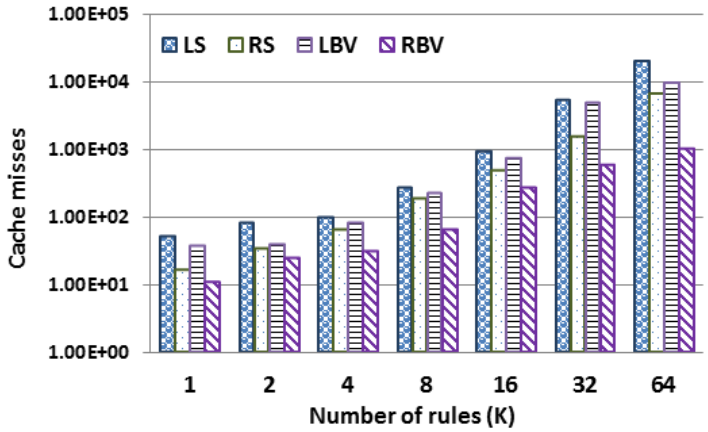
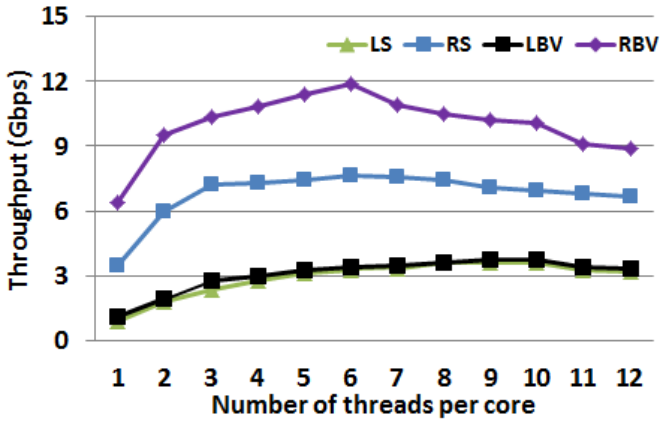


Figure 7. Number of L2 cache misses per 1K packets on the AMD processor

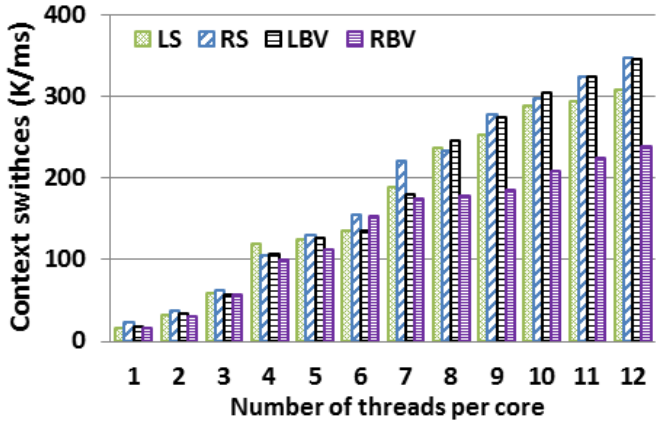
To explore further why the performance deteriorates as the rule size grows, we measure the cache performance. We show the number of cache misses per 1K packets under various scenarios on the AMD processor in Figure 7. As can be seen:

- Linear search based approaches result in more cache misses than the range-tree search based approaches because the linear search requires all the unique rules to be compared against the packet header.
- Approaches based on set representation have more cache misses than the BV based approaches. Set representation requires log(N) bits for each rule ID, while the BV representation only requires 1 bit to indicate the match result for each rule. Hence BV representation of the matched rules can be fit in the cache of a multi-core processor more easily.
- The overall performance is consistent with the cache hits on the multi-core processors. RBV introduces the least amount of cache misses and achieves the highest overall throughput with minimum processing latency.

4.6 Number of Threads per Core (T)



(a) Throughput vs. number of threads per core



(b) Context switch vs. number of threads per core

Figure 8. Performance vs. number of threads per core

The number of threads per core also has an impact on the performance for all four approaches. Our results, as shown in Figure 8a, indicate the performance of RS and RBV goes up as the number of threads per core (*T*) increases from 1 to 6. Once *T* exceeds 6 the throughput begins to degrade. For LS and LBV, the performance keeps increasing until *T* reaches 10. Reasons for performance degradation include saturated resource consumption of each core and the extra amount of overhead brought by the context switch mechanism. Figure 8b illustrates the relation between context switch frequency and the number of threads per core. When *T* increases, context switches happen more frequently, and switching from one thread to another requires a large amount of time to save and restore states.

We also evaluate $T > 13$ in our experiments, and we observe the performance drops dramatically for all four approaches. The performance of the approaches based on linear search deteriorates for $T > 10$, while the performance of range-tree based approaches deteriorates for $T > 6$. We explain the underlying reasons why the performance of range-tree based approaches degrades at a smaller T as follows:

- The different threads in the same core may go through different paths from the root to the leaf nodes, when range-tree search is performed. For a large number of threads in the same core, cache data cannot be efficient shared among different threads; this leads to performance degradation at a smaller value of T.
- For linear search, since all the threads use the same data and have regular memory access patterns in the search phase, the common cache data can be efficiently shared among different threads. However, as can be seen from Figure 8b, the frequent context switches still adversely affect the overall throughput performance. This in turn results in performance degradation at a larger value of T.

5 Conclusion and Future Work

We implemented and compared four decomposition-based packet classification algorithms on state-of-the-art multi-core processors. We explored the impact of the rule set size (N), the number of threads per core (T), and the communication between cores on performance with respect to latency and throughput.

We also examined the cache performance and conducted experiments on different platforms. Our experimental results show that range search is much faster than linear search and is less influenced by the size of rule set. We also find that BV representation based approaches are more efficient than set representation based approaches.

In the future, we plan to apply our approaches on OpenFlow packet classification where more packet fields are required to be examined. We will also implement decision-tree based and hashed based packet classification algorithms on multi-core platforms.

References

1. J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow Switch on the NetFPGA Platform", in Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ser. ANCS '08, (2008) 1–9.
2. G. Brebner, "Softly Defined Networking", in Proc. of the 8th ACM/IEEE Symp. on Architectures for Networking and Communications Systems, ser. ANCS '12, (2012) 1–2.
3. F. Yu, R. H. Katz, and and T. V. Lakshman, "Efficient Multimatch Packet Classification and Lookup with TCAM", IEEE Micro, vol. 25, no. 1 (2005) 50-59.
4. W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: an SRAM based parallel multi-pipeline architecture for terabit IP lookup", in Proc. IEEE INFOCOM (2008) 1786-1794.
5. G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A Scalable High Throughput Firewall in FPGA", in Proc. of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), (2008) 802-807.

6. T. Koponen, “Software is the Future of Networking,” in Proc. of the 8th ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS), 2012, pp. 135–136.
7. “AMD Multi-Core Processors,” http://www.computerpoweruser.com/articles/archive/c0604/29c04/29c04.pdf.
8. “Intel Multi-Core Processors: Making the Move to Quad-Core and Beyond,” http://www.cse.ohio-state.edu/~panda/775/slides/intel_quad_core_06.pdf.
9. “Multicore Computing- the state of the art,” http://eprints.sics.se/3546/1/SMI-MulticoreReport-2008.pdf.
10. P. Gupta and N. McKeown, “ Packet classification on multiple fields”, In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, SIGCOMM (1999) 147-160.
11. W. Jiang and V. K. Prasanna, “A FPGA-based Parallel Architecture for Scalable High-Speed Packet Classification,” in 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), (2009) 24-31.
12. P. Gupta and N. McKeown, “Packet Classification using Hierarchical Intelligent Cuttings”, IEEE Symposium on High Performance Interconnects (HotI) (1999).
13. S. Singh, F. Baboescu, G. Varghese and J. Wang, “Packet Classification using Multidimensional Cutting”, ACM SIGCOMM (2003) 213-224.
14. D. Liu, B. Hua, X. Hu and X. Tang. “High-performance Packet Classification Algorithm for Any-core and Multithreaded Network Processor.” in Proc. CASES, (2006).
15. D. E. Taylor and J. S. Turner, “Scalable Packet Classification using Distributed Crossproducing of Field Labels,” in Proc. IEEE INFOCOM, (2005) 269–280.
16. P. Zhong, “An IPv6 Address Lookup Algorithm based on Recursive Balanced Multi-way Range Trees with Efficient Search and Update”, in Proc. of international conference on Computer Science and Service System (CSSS), ser. CSSS ’11, (2011) 2059–2063.
17. T. V. Lakshman, “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching”, ACM SIGCOMM (1998) 203-214.
18. F. Pong, N.-F. Tzeng, and N.-F. Tzeng, “HaRP: Rapid Packet Classification via Hashing Round-Down Prefixes”, IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 7, (2011) 1105 –1119.
19. W. Jiang and V. K. Prasanna, “Field-split Parallel Architecture for High Performance Muti-match Packet Classification using FPGAs," in Proc. of the 21st Annual Symp. on Parallelism in Algorithms and Arch. (SPAA), 2009, pp. 188-196.
20. V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, “Fast and Scalable Layer Four Switching," in Proc. ACM SIGCOMM, 1998, pp. 191-202.