POPE and PaNeL : Fast Lookup in Routing Tables

H. B. Acharya
acharya@cs.utexas.edu

***Abstract*—Resolution - deciding on the appropriate action for a packet, based on a provided table of rules - is an extremely important problem in computer networks. Both routing and filtering require fast resolution. In this paper, we explore the possibility of using XMT, a recently-implemented FPGA-based general parallel machine, to perform resolution. While XMT is still not mature, our early results show that our systems, POPE and PaNeL, outperform the state-of-the-art serial solution by one to two orders of magnitude. Thus, we suggest that XMT may become a viable solution for fast packet processing in the future.**

I. Introduction

An important primitive in the operation of packet-switched networks, such as the Internet, is to check and see which rule in a policy applies to a packet. Routers examine incoming packets and decide, based on the routing policy, what course of action to pursue for each packet. Another important example occurs in system security, where packets must be checked by Access Control Lists such as firewalls and Intrusion Detection Systems. This operation of checking which rule in the policy to execute for a packet (or more generally an event) is called *resolving* the packet. It is a well known bottleneck in many 'high velocity' systems - for example, at internet backbone routers, performance is limited by lookup [1], i.e. the time for the policy to resolve the packet. This has led to a considerable research effort to develop data structures and algorithms for fast resolution [2] [3] [4] [5].

In addition to the development of algorithms and data structures, the need for fast resolution has led to the use of specialized machines and architectures, such as Ternary Content-Addressable Memory [10]. These solutions, however, are both very expensive and limited in the size of the policies they can use. With the current rise of parallel computing platforms and architectures, it becomes reasonable to ask whether more general approaches to parallel computing can be useful as a solution for the fast resolution problem. In this paper, we explore the suitability of one such target platform, XMT (Extensible Multi Threading) [7]. XMT implements a standard model (an approximate version of the arbitrary concurrent-read, concurrent-write PRAM) and is thus simple to program from a given parallel algorithm description. Further, it is the first near-PRAM machine to be prototyped inexpensively, using FPGA [6], and an open source simulator is also available; thus, it is feasible to develop and execute software targeting it as a platform.

In this paper, we present two XMT-based systems to do fast resolution. Our first experimental attempt to develop a fast resolver based on XMT is the Packets on Parallel Engine (POPE) system, which gives very competitive results for small policies (≈ 1000 rules), but falls behind for larger policies. We solve this problem by developing Parallel Next-step Lookup, i.e. PaNeL, which turns resolution into a batch processing problem and uses the parallel XMT machine to process all the rules in a batch in parallel. PaNeL shows very good results for large policies - in our tests, we show that it provides very good performance even for the largest policies we can test, of up to 100000 rules.

In our studies comparing PaNeL and POPE with the serial resolution algorithm, we found that several factors besides the length of the policy (i.e. the number of rules) play a very important role in our results. This paper presents these factors - domain width and rule width - and suggest that future studies of routing and filtering policies, as well as of first-match rule sequences in general, may want to pay attention to these metrics as well as the sequence length.

We begin by laying out our definitions and concepts, and a brief description of XMT and PRAM, in the next section. After this, we discuss the POPE and PaNeL algorithms, and present our experimental results. Finally, we mention how our work fits into the context of related research in the area, and offer some concluding remarks.

II. Terms and Concepts

In this section, we present a list of terms and concepts used in the paper, and show how resolution can be reduced to a simple model. We also introduce the XMT machine.

A. Packets, Rules and Resolution

A *packet* has various attributes which determine what action the policy takes when it arrives; these attributes are called *fields*. We model a packet as a d-tuple of non-negative integers, representing the d fields examined.

A *rule* R consists of a *predicate* and a *decision*.

The rule predicate is of the form

$$x_1 \in [R.x_{1,1}, R.x_{1,2}] \ldots x_d \in [R.x_{d,1}, R.x_{d,2}]$$

Each interval $[R.x_{k,1}, R.x_{k,2}]$ is an interval of non negative integers, drawn from the domain of field k. (For example, suppose the third field in packets and rules represents source IP address. In IPv4, the domain of this field is $[0, 2^{32} - 1]$. Then, in any rule R, $0 \leq R.x_{3,1} \leq R.x_{3,2} \leq 2^{32} - 1$. The domain width of this field is 32 bits.)

The decision is an action, such as the name of the interface to forward the packet out on.

A packet that satisfies the predicate of a rule is said to *match* the rule. For example, the packet $(1, 26, 7)$ clearly matches the rule

$$x_1 \in [0, 108] \wedge x_2 \in [21, 65535] \wedge x_3 \in [7, 616] \rightarrow accept$$

978-1-4799-3088-3/14/$31.00 ©2014 IEEE

A rule that cannot be matched by any packet is called a *match-none* rule, while a rule matched by every possible packet (in the domain of packets under consideration) is called a *match-all* rule.

A *policy* consists of multiple rules, and specifies their precedence (i.e. a way to decide which rule's decision is executed when multiple rules match a packet). For completeness, we enumerate the standard precedence algorithm for routers below:

1) Pick one field (usually destination IP). The rule which has the smallest interval for this field, and is still matched by the packet, wins. (Best match.)
2) In case of a tie, choose rules in the order:
   a) Static routes
   b) Dynamic routes, in order (usually EIGRP, OSPF, ISIS, RIP)
   c) Default route
3) If no rule matches, discard the packet.

In our model, we reduce the precedence algorithm to 'first matching rule wins'. [It may be noted that the two algorithms have equivalent power. To convert a practical policy to have first match semantics, we place the more specific before the less specific rules, and in case of ties, place the rules in the order: static rules, then dynamic, then default rules.]

The 'winning' rule, the decision of which is implemented for a packet, is said to *resolve* the packet.

Clearly, the naive approach of testing packets against rules serially, in decreasing order of precedence, has an expected resolution time of the order $O(nd)$, where n is the number of rules and d is the number of fields. (n can be quite large - 10^5 in a large policy or blocklist - though d is very small, about 5.) In our POPE and PaNeL systems, we improve upon this time bound using a standard parallel machine, XMT [7]; accordingly, we provide a brief description of XMT below.

B. Explicit Multi Threading (XMT)

XMT is an approximate implementation of a Parallel Random Access Machine (PRAM), i.e. a shared-memory machine model in which an unbounded number of processors have unit-time access to unbounded memory. (This is a theoretical extension of the standard RAM machine model, in which all memory accesses by a single processor take unit time.) PRAMs are classified by whether writes and reads to memory are exclusive or concurrent (i.e. whether multiple threads can write or read the same variable at the same time without waiting.) XMT attempts to implement an arbitrary CRCW PRAM; in other words, both reads and writes of any shared variable are concurrent, and it is arbitrary which processor 'wins', i.e. whose write gets recorded in case there is a conflict. We assume that the machine is *robust*, that is, an arbitrarily large number of threads can attempt to write the same variable without error. (In practice, XMT implements an intermediate between arbitrary CRCW and queued-read, queued-write, i.e. QRQW; however, following Vishkin et al, we assume it acts as an approximate PRAM.)

However, XMT has one primary limitation. Currently, it does not permit either function calls, or spawns of threads, while currently inside a spawn block. Hence, nested spawns are not permitted. If we consider a trivial PRAM algorithm,

```
1: for i ← 1, n do
2:     sum ← sum + i
3: end for
4: for j ← 1, m pardo
5:     for k ← 1, p pardo
6:         prod[j][k] ← j * k
7:     end for
8: end for
```

Using a standard **for** loop, the first n integers are added up to sum serially. The two nested loops make use of **pardo**, i.e. 'do in parallel'; the array $prod[m][p]$ is thus filled with the products of its indices in a single parallel step.[It may also be noted that for simple notation, we use arrays starting with position 1 rather than position 0.]

Such nested **pardo** statements cannot be translated into spawn blocks directly in XMT-C, but it is possible to convert this algorithm to run on XMT, without serializing either the outer or the inner loop, using the following transformation.

```
1: for i ← 1, n do
2:     sum ← sum + i
3: end for
4: for j ← 1, mp pardo
5:     k ← j mod p
6:     if k == 0 then
7:         k ← p
8:     end if
9:     prod[j/p][k] ← (j*k)/p
10: end for
```

This current limitation in the expressive power of XMT-C proved to be a problem in developing a parallel packet resolver: it is not, to the best of our knowledge, possible to wait on the event of receiving a packet and spawn a thread (or fork a process) to process it. However, we demonstrate in the next two sections that it is indeed still possible to do parallel packet processing on XMT, and to achieve remarkably good performance in packet resolution.

III. Fast Resolution

In this section, we begin by presenting POPE, our first parallel algorithm for packet resolution. Next, we discuss a weakness in POPE and present a new system, PaNeL, which overcomes this limitation.

The fundamental operation in packet resolution is testing whether the packet matches a given rule. This is a simple and fast operation: for every field, out of d fields, the value of the field for the packet is tested against the corresponding interval for the rule. It requires only two operations to check if this value is inside the corresponding interval (compare to start, compare to end). Hence each rule can be tested for a match in $O(d)$ time.

Given a parallel machine, there are clearly two methods of speeding up resolution by making use of parallelism. The first method is to parallelize over packets: we run a separate thread to process a new packet as soon as it arrives (without waiting for processing of earlier packets to finish). The second method

Fig. 1. The POPE algorithm for Packet Resolution

```
1: procedure PARALLELRESOLVE(Policy {R_1, R_2..R_n}, Packet p)
2:     A[n], A^+[n] : integer arrays
3:     ans ← −1 : default ans
4:     for i ← 1, n pardo
5:         A[i] ← Match(R_i, p)
6:     end for
7:     A^+ ← PrefixSum(A)
8:     for i ← 1, n pardo
9:         if A[i] == 1 ∧ A^+[i] == 1 then
10:            ans ← R_i.D
11:        end if
12:    end for
13:    return ans
14: end procedure
```

Fig. 2. The PaNeL algorithm for Packet Resolution

```
1: procedure PARALLELNEXTSTEPLOOKUP(Policy {R_1, R_2..R_n}, Packet p)
2:     step ← 1000, index, start, stop
3:     ans ← −1 : default ans
4:     for index ← 1, n/step do
5:         start ← (index − 1) ∗ step + 1
6:         stop ← min(start + step − 1, n)
7:         ans ← ParallelResolve({R_start, ...R_stop}, p)
8:         if ans ≠ −1 then
9:             break
10:        end if
11:    end for
12:    return ans
13: end procedure
```

involves parallelizing over rules: we check if a packet matches the rule, for all the rules in parallel.

Unfortunately, XMT is still in early development and does not support many important features such as the ability to run an OS, or multiple processes. The XMT model of parallel computing involves spawning a number of threads and waiting for them all to finish and join; once a spawn block has started, new threads are not supported. We have not so far been able to develop an solution that can dynamically spawn a new thread as soon as a packet arrives. Hence for the present paper, we focus on how to parallelize packet resolution over rules, rather than over packets.

Clearly, simply checking if a packet matches some rule out of all the rules is sufficient if a packet matches exactly one or zero rules, but not when it matches multiple rules. In case of multiple match, we require a method of deciding which rule has priority. As discussed in the previous section, we follow first-match semantics: the rules are ordered in a sequence, and the first rule in the sequence, that is matched by a packet, resolves it. So the question of fast resolution of packets reduces to the problem of finding the *first* rule matched by a packet.

The primary idea of the POPE algorithm is to find the first matching rule by using the concept of *prefix sum*. The prefix sum of an element $A[i]$ in an array A is the sum $A[1] + A[2] + ..A[i]$, and the prefix sum of the array A is the array A^+ such that for all i, $A^+[i]$ is the prefix sum of $A[i]$. Our key observation is that the first non zero element in a $(0/1)$ array is the only element which is 1 and whose prefix sum is also 1. (All the other elements either have the value 0, or if they contain 1, this 1 adds to the prefix sum so its value changes.) Computing the prefix sum for an array of n elements takes $O(\log n)$ time [8], and after this computation all the elements can be tested for this condition in parallel; the test requires $O(1)$ time.

The working of the POPE algorithm (Figure 1) is as follows. We store the results $(0/1)$ of whether the packet matches rule R_i in the policy Y into element $A[i]$ of array A. We then perform prefix-sum of A to get A^+, and finally resolve the packet with the decision of rule R_i iff $A[i] == A^+[i] == 1$. The total time to find the rule in a policy that resolves a packet is $O(d) + O(\log n) + O(1)$, i.e. $O(d + \log n)$.

However, this analysis assumes an infinite supply of parallel computing units (in this case, Thread Control Units, i.e. TCUs). In Section IV, we see that for our actual implementation of POPE, the parallel algorithm outperforms the serial version by an order of magnitude for small policies (1000 rules), but the gap begins to close as policies grow larger. The probable reason for this behavior is that in very large policies, the early rules resolve almost all the packets - the last rules only resolve the (very few) packets left unresolved by the (thousands of) rules above them. Consequently, the time taken by the serial algorithm becomes roughly stable for large policies. However, the time taken by POPE increases steadily, as it still has to process all the rules of the policy, and the number of available TCUs is limited (in our tests, it was the default 1024).

Based on this insight, we develop a new algorithm, PaNeL. PaNeL makes use of the available parallelism in an XMT configuration by reducing the policy to "batches" of rules. As there are ≈ 1000 processors, we call the same ParallelResolve function used in POPE with the given packet and a policy consisting of the first 1000 rules of the given policy Y. If no rule in the first 1000 resolves the given packet, we apply ParallelResolve again, this time with a policy consisting of the next 1000 rules, and so on. This continues until some rule resolves the packet, or there are no more rules in policy Y.

In the next section, we present experimental evidence that POPE works very well for small policies, and PaNeL solves the performance problem for large policies of size $10^4 - 10^5$ rules.

IV. EXPERIMENTAL RESULTS

In this section, we discuss the implementation of our systems PaNeL and POPE, and their time and performance characteristics.

XMT supports a dialect of C, named XMT-C, which is very similar to ordinary C. The additional function **spawn**(a, b), which creates threads with thread ID from a to b, is accessed by including the header file

```
<xmtc.h>
```

The execution of the parallel code, being performed by

Fig. 3. Resolution time: Interval-rule Policies

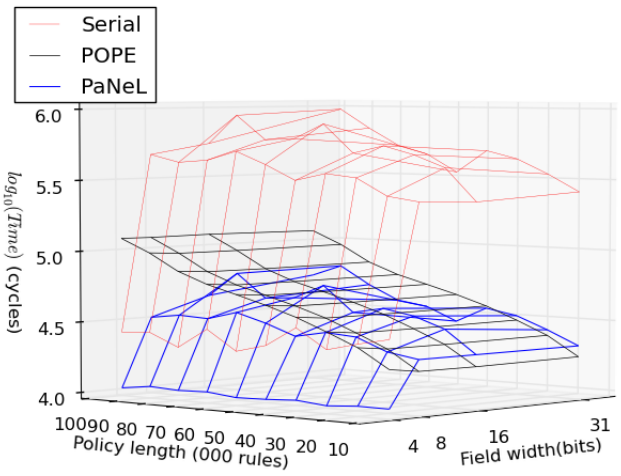
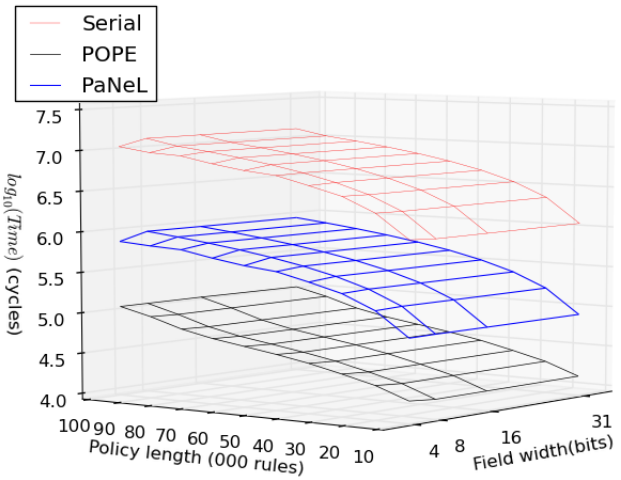


Fig. 4. Resolution time: Singleton-rule Policies



a simulator on a serial machine, becomes serialized; consequently, it is not useful to measure the actual time for execution. For timing, we use the built-in simulator macro, **xmt_readtimer32**(x), which when called loads the (32-bit) cycle count into (integer) x.

We also report a surprising finding. In our experiments, we first ran a small simulation using policies in which the domains of the fields were restricted to be $[0, 15]$, i.e. 4-bit, before the 'full' simulation with 31-bit fields. To our surprise, the two simulations yielded dramatically different results. In this section, we also report the results of our experiments to see if this metric (which we call 'domain width' of the fields in rules) is an important metric of the complexity of policies. [This would be very important as, with the shift to IPv6, policies will have to change from 32-bit to 128-bit IP domains.] We demonstrate the results of POPE and PaNeL with different domain widths.

A. Experiment 1: Interval-based rules

In our first experiment, we measure the average time (over 100 packets) required for packet resolution by random policies of length varying from 10000 to 100000 rules. The rules in our policies have five fields; in order to measure the impact of domain width, we vary the width of the fields from 4 to 31 bits (i.e. the field domain varies from $[0, 15]$ to $[0, 2^{31} - 1]$). [We attempted to use larger widths up to 128 bits, but our tests indicated that 32 bits is a current limitation for the XMT-C compiler.]

Our results are presented in Figure 3. As can be seen, the resolution time drops sharply for very small domains (4 bits) in both the serial and the PaNeL algorithms, but not for POPE. However, from 8 to 31 bits there is no clear trend. We conclude that for policies with interval-based rules, domain width is a significant factor only when very small; it seems very likely that algorithms that work well for IPv4 policies (where the domain of a field is up to 32 bits) should also work well with IPv6 (where the domain can go up to 128 bits).

Our test results show that PaNeL is a little faster than POPE for policies of up to about 50000 to 60000 rules, and then pulls ahead. By a policy length of 100000 rules, PaNeL is more than twice as fast as POPE, and over 20 times faster than the serial solution.

B. Experiment 2: Singleton rules

In our second experiment, we study the impact of changing the width of the fields in the rules themselves (rather than the width of the domains of the fields). As our rules have random intervals in their fields, it is not possible to simply mandate narrower intervals (other than by narrowing the domain). To solve this problem, we came up with the concept of *singleton* rules. In contrast to interval-based rules, in which the predicate is of the form

$$x_1 \in [R.x_{1,1}, R.x_{1,2}] \ldots x_d \in [R.x_{d,1}, R.x_{d,2}]$$

we constructed policies of singleton rules, which have predicates of the form

$$x_1 \in [R.x_{1,1}, R.x_{1,1}] \ldots x_d \in [R.x_{d,1}, R.x_{d,1}]$$

In other words, for every field in a singleton rule, there is exactly one value, rather than an interval of values; the corresponding field of a matching packet must have exactly this value. [We note in passing that such rules are quite common in firewalls; most or all the fields in a firewall rule are frequently a single value only.]

The last rule of the policy, a match-all default rule, is the only non-singleton rule. The other details of the experiment are identical to Experiment 1.

Our results are presented in Figure 4. Notably, the performance of all three algorithms shows very similar characteristics, and varies little with domain width (though there is still a small dip for very small domains of 4 bits). However, while PaNeL is one order of magnitude faster than the serial algorithm, POPE is one order of magnitude faster still. We conclude that for large policies of singleton rules, POPE is still the better algorithm.

We provide a clear comparison of the time required by the serial, POPE, and PaNeL algorithms for packet resolution in

Fig. 5. Serial Resolution time

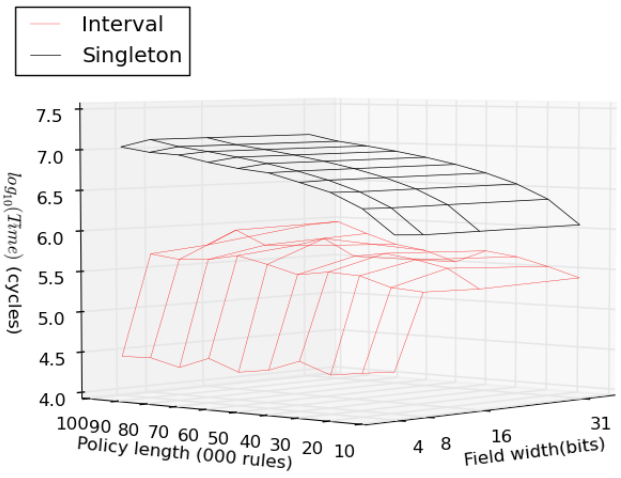
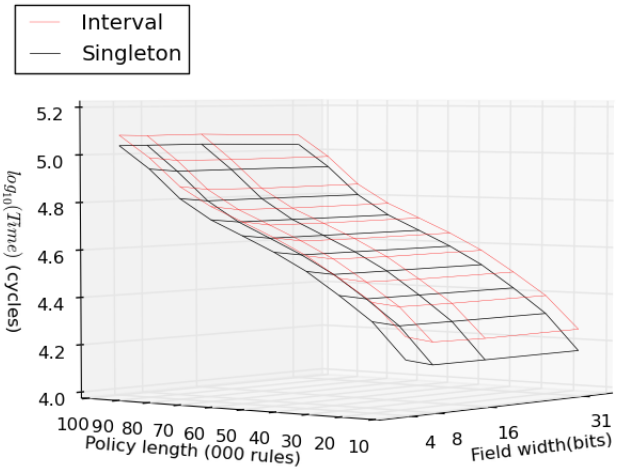


Fig. 6. POPE Resolution time



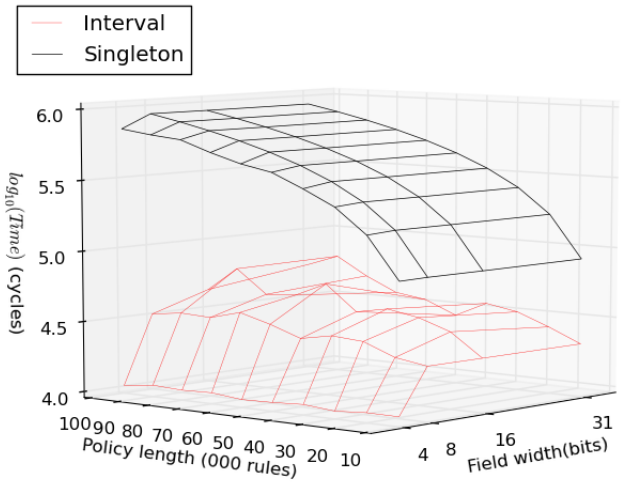
case of policies with interval-based and singleton-based rules in Figures 5, 6, and 7.

While both the serial and the PaNeL resolution algorithms show a very similar performance trend (high, predictable time for singleton based and much lower, more irregular time for interval based rule policies), POPE shows almost the same performance for both types of policy. This is as expected: POPE has to do the same amount of work in both cases. It may also be noted that the shape of the PaNeL and the serial graphs are very similar. This is because the same test packets are used in all cases, and in general packets which are resolved by a later rule take more time in both PaNeL and in serial resolution.

V. Related Work

Our work in this paper contributes to the study of fast resolution. Most of the current research in this area has studied the use of advanced data structures such as tries [9], but the underlying architecture is a simple serial machine. Lookup-table based solutions, as studied by Waldvogel [2] and Gupta

Fig. 7. PaNeL Resolution time



[3], offer very good read performance - $O(x+\log n)$ and $O(1)$ respectively - but the second approach requires impractical amounts of memory. Such solutions are also limited by the fact that precomputation of the data structure is expensive. To make dynamic updates feasible, Suri [4] proposes using B-trees, and Sahni and Kim use red-black trees and skip lists [5]. Using these data structures, longest-prefix matching on a single field can be performed in $O(\log n)$ time. Also, the B-tree solution allows policies to be updated in $O(x+\log n)$ expected time, and the collection of red-black trees, in $O(\log n)$. (x is the width of the field of the new rule.) However, these solutions are usually only applicable to rules with only one field. Further, the field must be specified in the form of a prefix, or only non-overlapping ranges.

Our work on fast resolution, as discussed in Section III, is complementary to these techniques; it falls in the field of using high-performance architecture. Dedicated high-performance solutions such as ternary content addressable memory [10] can reach $O(1)$-time resolution, but such solutions are extremely limited in the size and complexity of the policies they can hold, and expensive as well. In our work, we attempt to bring together the need for a fast packet resolver with current research in parallel computing; our POPE and PaNeL systems are built on XMT [7], a general machine for fine-grained massively parallel execution ("desktop supercomputer"). Our results show that despite its limitations, XMT shows promise in the field of packet resolution, and as it is a cheap general-purpose FPGA solution, it may become a viable option for fast packet processing in future. In our future work, we intend to explore the economic aspects of using an XMT solution, the ability to use XMT as a co-processor, and so on.

We believe that our work immediately suggests several possible avenues of future research. First, while XMT is the first usable implementation of its kind (an arbitrary CRCW PRAM-like machine), there are many competing systems for parallel programming, ranging from Cilk [11] to Galois [12] to general-purpose GPU [13]. These systems should also be investigated to check whether they can provide good performance for resolution. Secondly, our POPE and PaNeL systems

achieve fast resolution using a trivially simple representation of a policy (as an array of rules); it may be possible to do still better by combining a parallel machine with a clever policy representation, as in Sahni et al [5]. There still remain some challenges in using XMT, such as how to develop a system that parallelizes over packets (i.e. uses the available parallelism to process multiple packets in parallel). Finally, it would be interesting to see if our model of rules and packets can be made stronger, such as by allowing rules that recognize regular expressions; this forms the basis of our own current research.

It may be noted that besides its immediate applications in improving the performance of large policies, our work has implications for policy verification [14], optimization [15] and testing [16]. For example, in working with the system Probe [17], a state of the art algorithm for verification of firewalls, we note that about $80 - 90\%$ of the time for verification is spent resolving packets! Similarly, in testing [18], the tester generates many packets, determines the "expected" decisions of the policy, and observes its actual decisions of for these packets. (If the expected decision for each generated packet is the same as the actual decision for the packet, one concludes that the given policy is correct; otherwise, it has errors.) It is natural to conclude that work on fast resolution can help in better design, analysis, and checking of policies. Further, we suggest that the new metrics we mention in this paper - domain width and rule width - have some impact in predicting performance, and intend to devote further study to the importance of these metrics in routing tables and firewalls.

VI. Conclusion

High-speed resolution, such as for fast routing of packets, is a well-studied problem, but many of the current architecture-based solutions are of limited use, as special-purpose hardware such as TCAMs are expensive. In this work, we study how a general purpose chip multiprocessor called XMT can be used to resolve incoming packets both quickly and correctly. We develop and present two systems.

Our first system, POPE, can resolve packets an order of magnitude faster than standard serial solution; however, for very large policies ($> 10^4$ rules) the performance gap begins to narrow. This is because in very large policies, the later rules are rarely used. Using this insight, we develop the second system, PaNeL, which not only clearly outperforms POPE for the usual case (interval based rules), but shows no signs of losing its performance advantage for very large policies.

During our work comparing POPE, PaNeL, and serial resolution, we realized that other factors besides the policy length can play an important role in the time taken by a policy to resolve a packet. Two such factors are the 'rule width' (whether a field in a rule has a single or multiple values) and 'domain width' (the number of bits used to specify each field in a rule). The power of rule width can be seen from the fact that, though PaNeL clearly outperforms POPE for interval based rules, in the case of singleton rules (where the fields are constrained to be a single value instead of an interval), POPE is the better solution. In practice, it seems likely that PaNeL would be better for a routing table, but POPE might be better for a firewall.

We conclude from our results that XMT is quite a good solution for developing a fast packet-resolving engine, and intend to explore the impact of combining XMT with fast algorithms and data structures for resolution in our future work.

References

[1] S. Keshav and R. Sharma, "Issues and trends in router design," *IEEE Communications Magazine*, vol. 36, pp. 144–151, 1998.

[2] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed ip routing lookups," in *Proceedings of ACM SIGCOMM*, 1997, p. 2536.

[3] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings of IEEE INFOCOM*, 1998.

[4] S. Suri, G. Varghese, and P. Warkhede, "Multiway range trees: Scalable ip lookup with fast updates," in *GLOBECOM*, 2001.

[5] S. Sahni and K. Kim, "O(log n) dynamic packet routing," in *IEEE Symposium on Computers and Communications*, 2002.

[6] U. Vishkin, "Algorithmic approach to designing an easy-to-program system: Can it lead to a hw-enhanced programmer's workflow add-on?" in *ICCD*, 2009, pp. 60–63.

[7] X. Wen and U. Vishkin, "Pram-on-chip: first commitment to silicon," in *SPAA*, 2007, pp. 301–302.

[8] G. E. Blelloch, "Prefix sums and their applications," Synthesis of Parallel Algorithms, Tech. Rep., 1990.

[9] K. Sklower, "A tree-based routing table for berkeley unix," in *Technical report, University of California*, 1993.

[10] D. Shah and P. Gupta, "Fast updating algorithms for tcams," *IEEE MICRO*, vol. 21, no. 1, p. 3647, 2001.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995.

[12] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. hsien Lee, A. Lenharth, R. Manevich, M. Mndez-lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of PLDI*, 2011, pp. 12–25.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," 2007.

[14] H. B. Acharya and M. G. Gouda, "Linear-time verification of firewalls," in *Proceedings of the International Conference on Network Protocols*, 2009.

[15] ——, "Firewall verification and redundancy checking are equivalent," in *INFOCOM*, 2011, pp. 2123–2128.

[16] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, 2001, pp. 308–316.

[17] H. B. Acharya and M. G. Gouda, "Projection and division: Linear-space verification of firewalls," *Distributed Computing Systems, International Conference on*, pp. 736–743, 2010.

[18] D. Hoffman and K. Yoo, "Blowtorch: a framework for firewall test automation," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 96–103.