A Performance Model for Automatic Optimization of Software Packet Processing Pipelines

Author 1, Author 2, Author 3, and Author 4
Affiliation
{author1,author2,author3,author4}@affiliation.org

ABSTRACT

Software packet processing is increasingly commonplace, especially for software-defined networking constructs. Previous work has investigated methods to efficiently map packet processing pipelines to general-purpose processor architectures. Concurrently, novel high-level domain-specific languages (DSLs) for specifying modern packet processing pipeline functionality, are emerging (e.g., P4 [?]). An attractive goal is develop a compiler that can automatically map a high-level pipeline specification (specified in a high-level DSL) to an underlying machine architecture. Ideally, the compiler should automatically exploit the available parallelism, make intelligent scheduling decisions, and adapt to the workload needs in an online fashion, to provide maximum performance. An important pre-requisite for the development of such a compiler, is a performance model of the underlying machine architecture, for the applications of interest.

We report our experiences with adding an optimizer to the P4C compiler [13], which compiles a high-level P4 program to a lower-level C-based implementation that runs with the DPDK infrastructure [1], and gets eventually executed on a multi-socket x86 machine. We make two contributions: (a) we show that significant performance improvements (up to 55%) can be gained by adding scheduling and prefetching optimizations to the P4C compiler; and (b) we develop a performance model for reasoning about the expected throughput and latency of a packet-processing workload, on a modern machine architecture. Our model can be used by a compiler, to reason about the expected performance of a packet-processing workload for different code configurations, and can thus be used to optimize the generated code accordingly.

CCS CONCEPTS

• **Networks** → **Network performance analysis**; **Programmable networks**;

KEYWORDS

Software Switch, Batching, Prefetching

ACM Reference format:
Author 1, Author 2, Author 3, and Author 4. 2017. A Performance Model for Automatic Optimization of Software Packet Processing Pipelines. In *Proceedings of ACM SIGOPS Asia-Pacific Workshop on Systems , Mumbai, India, September 2-3, 2017 (APSys'2017),* 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

APSys'2017, September 2-3, 2017, Mumbai, India

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software defined networking is already mainstream. Newer protocols and innovative packet processing functionalities, imply that newer packet processing pipelines are getting developed over time. Manually optimizing such packet processing pipelines requires highly-skilled programmers, is tedious and time-consuming, and can be repititive. For example, the RouteBricks paper [?] discusses several insightful optimizations for a simple packet-forwarding application on a modern multi-core machine; doing such careful research for each separate (and potentially much more complex) packet processing pipeline seems impractical. Further, these optimizations need to be re-tuned for every different architecture.

Domain specific languages like P4[5] and Click[12], are intended to bridge this gap, by allowing manual specification of functionality using high-level constructs. In this way, several low-level details are abstracted away from the programmer. However, the onus of efficiently mapping this high-level specification to the underlying machine architecture, shifts to the compiler. The difference between an unoptimized and optimized implementation for the same high-level specification, can be significant. For example, Kim et. al. [11] optimize the specification of an an IPv4 forwarding engine, specified using the Click programming model, to achieve 28 Gbps of IPv4 throughput for minimum-sized packets on a machine with two quad-core Intel Xeon X5550 CPUs, i.e., they achieve roughly 3.5 Gbps per CPU core for this workload. Similarly, P4C [?], a prototype compiler for P4, is able to achieve 5.17 Gbps IPv4 throughput per CPU core for an identical workload configuration, on an Intel Xeon E5-2630 CPU. In contrast, a hand optimized implementation for this workload can achieve over 10Gbps per core on an identical machine. An ideal compiler should be able to bridge this performance gap between compiler-generated code and hand-optimized code.

We evaluate two important compiler optimizations in this context, and evaluate their performance effects on common packet-processing workloads running on a commodity server machine: (1) efficiently exploiting the DMA bandwidth between NIC and main memory; and (2) efficiently exploiting the memory-level parallelism between CPU and main memory. For the latter, we also evaluate prefetching to minimize cache-miss stalls. Finally, we rely on existing C compilers (e.g., gcc) to automatically optimize the generated code to efficiently utilize the CPU processor (e.g., by maximizing SIMD and instruction-level parallelism ILP). We experiment with several code configurations involving varying degrees of available parallelism to DMA channels, and to CPU-memory channels. We show improvements over previously-published performance results for similar programs, and develop a model for reasoning about performance in this setting. We have integrated our model into the P4C compiler, to enable it to automatically optimize the example programs discussed in this paper.

2 AN ILLUSTRATIVE EXAMPLE

We illustrate the tradeoffs involved in implementing a packet-processing pipeline, using a motivating example of the L2 packet forwarding application. At the high-level specification, the application involves input streams of packets (one stream per input NIC) converging to a processing node that looks up the packets' MAC addresses (source and destination) to index locally-stored lookup tables, to decide the destination of the packet, and finally output streams of packets (one stream per output NIC) emerging out of the processing node. We use this simple application to illustrate the performance implications of scheduling and prefetching decisions made by the compiler. We first focus on performance optimization for a single CPU core. Multi-queueing support in modern NICs ensures that most applications can scale near-linearly with the number of cores.

We first discuss the characteristics of the underlying hardware. Three components of a general-purpose architecture, are most critical to the performance of the application: NIC subsystem, CPU, and Memory subsystem. Figure **??** illustrates the data-flow at the hardware level. The NIC is involved in reading the packets off the wire and storing them into main memory. In our experiments, a NIC's input bandwidth is upper-bounded by 10Gbps, however, its bandwidth to memory is usually much higher[1]. The latency from NIC to main memory is usually long (i.e., the NIC-memory interface is a high-bandwidth high-latency interface), and so it is important to allow multiple packets in flight from NIC to memory, to ensure effective bandwidth utilization across the NIC-memory interface. To realize this parallelism, we need *batching*, by ensuring that multiple packets are consumed/produced to the NIC's ring buffer in one shot. This allow the NIC to initiate multiple PCIe transactions in parallel, thus hiding the round-trip NIC-memory latency. This available parallelism at the NIC-memory interface, is labeled as (1) in Figure **??**.

The next step in each packet processing pipeline is the actual logic; this logic is usually the key differentiator across multiple applications. Some applications are CPU-bound (involve significant processing time), and others are memory-bound (involve significant cache-misses and round trips to main memory). If an application is CPU bound, we cannot do much, and rely on the underlying C compiler to tighten the code, and extract the SIMD and ILP parallelism. If the application is memory-bound however, our code generator can exploit the available memory-level parallelism.

The CPU-memory interface is also a high-bandwidth high-latency interface. CPUs allow multiple memory requests to be in flight, by using MSHRs (miss-status handling registers) to store the state of in-flight memory requests. Previous work on comparing CPU and GPU performance for packet-processing pipelines [?] highlighted the importance of exploiting memory-level parallelism in these workloads. An out-of-order superscalar CPU executes a *window* of instructions in parallel. Thus, a CPU can issue multiple main-memory requests in parallel, only if the consecutive memory requests happen to be within a single instruction window. Kalia et. al. [?] achieve this by *statically context-switching* among multiple threads, on each expensive memory access. They relied on the programmer to manually annotate the expensive memory accesses (the ones that are likely to result in a cache-miss) by hand.

We show that the memory-level parallelism can be exploited through *sub-batching* (a sub-batch is created within a larger batch that was required to efficiently NIC-memory bandwidth), for this CPU-memory interface. Sub-batching involves processing multiple packets (of sub-batch size) at each step of the processing logic. Sub-batching ensures that multiple independent lookups (if any) can be close-enough, such that memory-level parallelism gets exploited. Both batching and sub-batching, can also be thought-of as the loop-fission transformation in classical compiler literature [?], as are shown in Figures 1 and 2. We use B to denote the batchsize, and b to denote the sub-batchsize.

```
sub app {
  for (i = 0; i < B; i++) {
    p = read_from_input_NIC();
    p = process_packet(p);
    write_to_output_NIC(p);
  }
```

```
sub app {
  for (i = 0; i < B; i++)
    p[i] = read_from_input_NIC();
  for (i = 0; i < B; i++)
    p[i] = process_packet(p[i]);
  for (i = 0; i < B; i++)
    write_to_output_NIC(p[i]);
}
```

Figure 1: Batching, shown as loop-fission compiler tranformation.

```
sub process_packet(p) {
  for (i = 0; i < B; i++) {
  t1 = lookup_table1(p[i]);
  t2 = lookup_table2(p[i], t1);
  ...
}
```

```
sub process_packet(p) {
  for (i = 0; i < B; i+=b) {
  for (j = i; j < i+b; j++)
    t1[j-i] = lookup_table1(p[j]);
  for (j = i; j < i+b; j++)
    t1[j-i] = lookup_table2(p[j], t1[j-i]);
  ...
}
```

Figure 2: Sub-batching, within `process_packet` **shown as loop-fission compiler tranformation.**

Sub-batching is best performed through the *prefetch* instruction available on the x86 architecture. The prefetch intruction tells the hardware that this memory request is less critical than the other memory requests, allowing better scheduling at the hardware level. Algorithm **??** shows the example prefetching code used for the hash-lookup in the L2 forwarding example.

[1]The total PCIe bandwidth to main memory is around XXX, which gets shared across multiple NICs. For all our experiments, except one (XXX), the total PCIe bandwidth remains unsaturated.

Algorithm 1 HASH LOOKUP

```
for i ← 1 To BatchSize do
    key-hash[i] = extract key and compute hash;        ▷ C_K
    prefetch(bucket-for-key-hash(key-hash[i]));        ▷ C_P
end for

for j ← 1 To BatchSize do
    value[j] = hash-lookup(key-hash[j]);               ▷ C_L
end for
```

Notice that sub-batching is dependent upon batching, in that, the sub-batchsize can only be smaller than the batchsize. Thus, if there is no batching, there can be no sub-batching. We experimentally show that the optimal sub-batchsize for a given batchsize, is often less than the batchsize.

Finally, the processed packets are transmitted to the output NICs. Assuming uniform distribution of output packets across output NICs, we expect the output NICs to automatically operate at batch granularity. Hence, the output bandwidth utilization at the memory-NIC interface is similar to the utilization of the input NIC-memory utilization.

3 EVALUATION

Evaluation section is divided into four types of experiments: First, we are showing the effect of batching and batching plus prefetching for each application. Second, we are comparing our applications with the same application with hand-tuned optimizations published in different papers. By doing this we are showing that automatic optimizations can work as good as hand-tuned optimizations. Third, we are running the applications with different number of cores to show that the applications are scaling linearly with the number of cores. Fourth, we are comparing the throughput by changing the batch size and table size for L2FWD application to show the relation among batch size, table size, and throughput. With this experiment, we are also making sure that our model is working the same way we are expecting.

3.1 Evaluation Setup

Server Specifications: We are using Dell Poweredge R430 Rack Server, based on Haswell architecture. This server has two sockets occupied with Intel Xeon E5-2640 v3[2] processor. Each processor has 8 physical and each core is capable of running at 2.60 GHz. Cores on a socket share 20 MB cache. Sockets are connected with 2 QPIs(Quick Path Interconnect), each capable of 8 GT/s. Two dual port NICs,1 Intel x520 and 1 Intel x540, are connected to Socket 0 through PCIe 2.0 and each port can work at 10Gbps. Total main memory available is 64 GB, spread across two sockets in a NUMA fashion.

Software: Our servers are running Ubuntu 14.04 LTS operating system with 4.4.0-59 Linux kernel version. We are using DPDK version 16.07 with IXGBE poll mode driver to interact with the underlying NICs.

Traffic Generator: We are using same hardware and software on both the servers and Pktgen-DPDK[4] version 3.1.0 to generate different kind of packets for different applications used in the experiments. Pktgen-DPDK[4] can generate the 64 bytes packet size traffic at line rate i.e 14.8 Mpps for 10 Gbps port. We are able to generate traffic at 59 Mpps for four ports with 64 bytes packet size. We have extended Pktgen-DPDK[4] to put random source and destination address depending on the application.

Applications: In this part we are specifying the applications we are using for different experiments. We have written applications in P4[5] and extended P4C[13] compiler to generate code with the above mentioned optimizations. P4C[13] is generating DPDK[1] based applications which are later compiled with gcc to get the target binary. We are using similar applications as used in [10] for comparing our results with their CPU based implementation.

(1) **Layer 2 Switch:** In this application we are using two hash tables to store the mapping between SMAC/DMAC and value. Each packet is going through two lookup stages, one for SMAC and one for DMAC. We are putting 16 Million entries in the table for the experiments unless otherwise specified for some experiment. We are using Intel DPDK's implementation for various hash table operations.

(2) **IPV4 Forwarding:** In this application we are performing LPM lookup on destination IP address to get the forwarding port. We are using Intel DPDK's implementation for LPM related operations for IPv4 Forwarding and IPv6 Forwarding application. We are populating the forwarding table with 527,961 prefixed used by [10].

(3) **IPv6 Forwarding:** We are performing one lookup on destination address to find the egress port. We are populating the DPDK LPM table with random 200,000 entries with the length between 48 to 64, as done in [10]. From the Pktgen we are generating the packets with destination address randomly picked from these 200,000 entries. Minimum packet size for this application is 78 bytes and not 64 bytes.

(4) **Named Data networking:** We are using hashtable for lookups implemented by DPDK. We are using algorithm and URL dataset from [14]. From Pktgen we are generating packets by randomly putting the URLs from used dataset. We are using 32 bytes URLs in the packet headers as done in [10].

(5) **l2fwd-crypto Application:** All other applications are lookup based and we want to see the effect of batching & prefetching on other kinds of applications too. So, we have included this application in the experiments. We are using L2Fwd-Crypto[3] from DPDK examples given in DPDK repository. The application performs encryption and decryption based on the input parameters and then it forwards the packet on Layer 2 with static port mapping.

3.2 Effect of batching and prefetching

In P4C[13], authors are not using batching and prefetching for the applications. We are adding batching and prefetching for all the applications and in Figure 3 we are showing the effect of these optimizations for different applications. We are using 32 as batch size for all the applications in all the experiments unless specified otherwise. L2Fwd and NDN are using hash lookup and DPDK code is written in such a way that we are able to prefetch the bucket where the key is present. On the other hand, we don't have much opportunity to use prefetches for LPM lookup based applications. For L2Fwd, batching improves the performance by 20% and prefetching, used

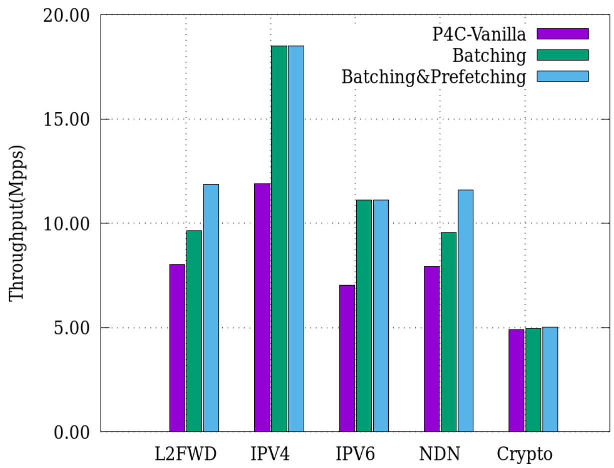

Figure 3: Effect of batching and prefetching

with batching, again improves the performance by 23%. Similarly for NDN, batching alone improves the performance by 20% and after adding prefetching there is an additional performance gain of 21 %. After applying batching and prefetching to IPv4 and IPv6 there is a performance gain of 55% & 57% respectively.

Conclusion: As we can see that batching can be used for almost all the applications without thinking much. The only challenge is to come up with the optimal batch size. Section **??** can be used to find the optimal batch size. Prefetch on the other hand surely improves the performance but can't be used in every application and should be used with precaution as it might pollute the cache which might result in performance loss.

3.3 Comparison with hand-tuned applications

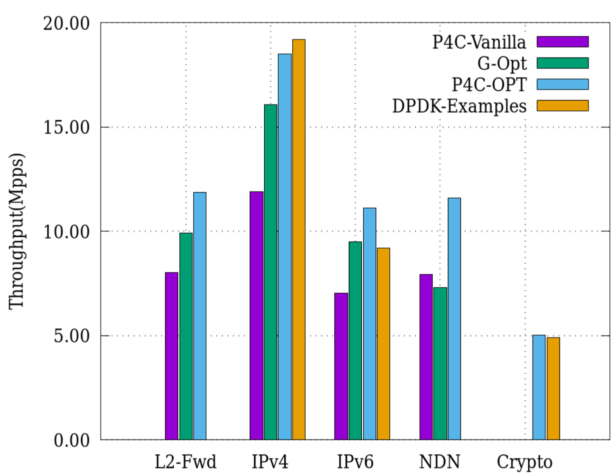

Figure 4: Comparison with other hand-tuned applications

In Figure 4, we are comparing our automatic generated code with comparable hand-tuned code and vanilla P4C[13]. We are making sure that the number of lookups and number of entries in the table(s) is same in all the cases so that we can show the comparison among applications. For some of the applications, it is not possible to compare our application with the hand-tunes application and we have omitted the bar from the graph for such applications.

Comparison with vanilla P4C[13]: This is the most suitable comparison because we are using the same applications and same flow in our applications. There is 48%, 55%, 57%, and 46% improvement for L2Fwd, IPv4, IPv6, and NDN application respectively. In the applications, they are not exploiting batching & prefetching and just with these optimizations there is a huge improvement.

Comparison with G-Opt[10]: There is 20%, 15%, 17%, 59% improvement for L2Fwd, IPv4, IPv6, and NDN application respectively. They are using both batching and prefetching, and their main aim was to make the results comparable to GPU. The difference is due to the batch size, they are not using the optimal batch size and we can see the performance gain by using the optimal batching size in Figure 6.

Comparison with DPDK[1]: For IPv4 DPDK is performing 4% better than our code and for IPv6 we have a performance gain of 20%. The result for IPv6 doesn't include the improvement we are gaining by TRIE compression, that will be reported later in this section.

Conclusion: By using the optimal batch size and right prefetch distance we are performing almost equal or better than other hand-tuned optimized code. Even if the results are equal we can say that our approach is better than hand-tuned one, because it is one-time efforts and can be used for wide variety of applications. Results in figure 4 are for one core and we will show in Section 3.4 that our applications are scaling linearly with the number of cores.

3.4 Scalability

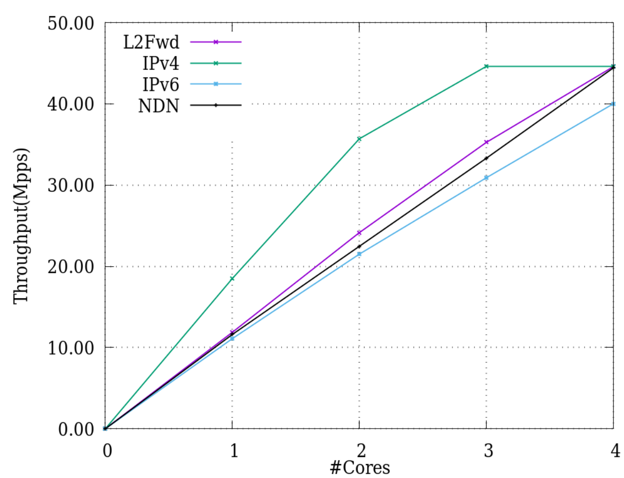

Figure 5: Number of Cores vs Throughput

In Figure 5, we are showing that our applications are scaling with the number of cores. With 4 cores we saturating 4 ports(4 x 10 Gbps) when the packet size is 64 bytes. The theoretical limit for 4 NICs is 59 Mpps but we are getting 44 Mpps as shown in the graph. PCIe is the bottleneck in this case and not the applications. When one NIC is sending and receiving at both the port then it can reach up 22 Mpps and not theoretical maximum 29 Mpps. [15] has also mentioned about this bottleneck in the paper.

There is not much to talk about L2Fwd and NDN application, these applications are scaling linearly. IPv6 is not reaching upto 44 Mpps with 4 cores and that is because of large packet size for IPv6. We are using 64 bytes packets for other applications but Pktgen-DPDK is generating 78 bytes packets for IPv6. Hence IPv6 application is also saturating the ports with 4 cores. The other interesting part of the graph is the IPv4 line, till 2 cores the application is scaling linearly and after that it is coming down. There is no more available bandwidth for the application and it is saturating it with three cores.

Conclusion: We can say two things about the result of this experiment. First, applications are scaling linearly with the number of cores and second, applications are saturating four 10 Gbps ports with four cores.

3.5 Relation between Batch Size, Table Size, and Throughput

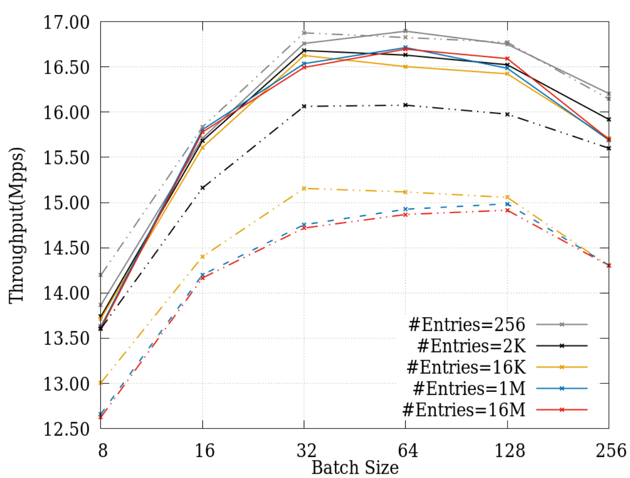

Figure 6: Throughput vs Batch Size

In Figure 6, we are showing the relation between throughput and batch size for different number of entries in the lookup table. We are using L2-FWD application with one lookup for this experiment. In the graph, solid lines are representing the results for batching plus prefetching case and dotted lines are representing batching only results. In Section 3.5.1 we will talk about the effect of batching on the application with varying number of entries in the table and in Section 3.5.2 we will take both batching and prefetching into consideration.

3.5.1 Effect of Batching. The throughput is decreasing with increase in the number of entries for the same batch size. This kind of behavior is expected and can be explained with the help of Eq. **??**. Eq. **??** and Eq. **??** tell that with the help of batching we can amortize per packet cost and the throughput of the application is improved. There is an improvement in the throughput until we are increasing the batch size from 1 to optimal batch size and after that, there is a decline in the throughput.

Batch Size vs Throughput: Optimal batch size is different for different cases and can be explained individually for each case. When tables size is small and can fit in L1/L2/L3, the optimal batch size is 32 and if we are increasing the batch size more than 32 then there is a small dip in performance because of the increased cache contention due to increased batch size. However when the table size is bigger than the LLC, for every packet the fetch time is constant and we can increase the batch size to get the benefits of batching. In case of large table size, the optimal batch size is 128 in our case.

Table Size vs Throughput: As mentioned in the eq **??**, memory stall is one of the parameters which is affecting the application throughput. Contention for cache will increase with the increase in the number of entries in the lookup table and this will increase the fetch time for lookup data. So, total fetch time will be more for large table size and less for smaller table size. As explained in Eq **??**, we can hide the memory stalling by prefetching the data in the cache. If we can prefetch the data efficiently before the data is used by the application then the total stall time for the batch would decrease and performance of the application would increase.

3.5.2 Effect of Batching plus Prefetching. With prefetching, we are trying to minimize the total stall time for the batch and we can see in the figure that there is a significant performance gain if we are using prefetching with batching. We can summarize the result in two main points. First, there should be a decline in the throughput with the increase in the number of entries and throughput of the application should be more when there are less number of entries in the table. For larger table size, data won't be present in the cache and application must stall on it. However, we are prefetching the data even before the data is used in the application and this prefetching of data is reducing the total stall cycles as described in Eq **??**. In case of batching plus prefetching both CPU and memory are working effectively and there are not stall in this case and this is the reason that the throughput of the application is stable and not varying much with the increase in the number of entries in the table.

Second, for each batch size, the relative throughput of the application will be dictated by the number of entries in the table. In actual, the results are quite opposite to the expectations and this is because total stall time reduction is dependent on effective prefetch distance. One batch size may not work for the different number of entries and relation between table size, batch size, and effective prefetch distance has been summarized in Section **??**. Let's explain with the help of an example, for 64 batch size the throughput of the application is more when there are 1 M and 16 M entries as compared to 2 K & 16 K entries in the table. The data will be in L2/L3 cache for these many numbers of entries and due to large batch size, entries will be prefetched way before the data will be needed in the application. On the other hand for larger table sizes, larger batch size is suitable since the data will be fetched from the main memory which will take more number of cycles. For 2 K and 16 K entry table we tried to minimize the prefetch distance by using the sub-batch size of 32 and the application is performing better than the application with 1 M and 16 M entries.

Conclusion: In case of batching only, the performance is decreasing with an increase in table size because of the increase in the stall time. We are using prefetching to minimize the stall time and in case of batching plus prefetching the throughput is not declining much due to increase in the table size. We can say that batching and prefetching is playing well together and due to this the throughput is stable even if we are increasing the table size.

3.6 TRIE Compression

In this experiment we enabled the DIR24-8[9] based DPDK LPM6 library to use TRIE compression. The number of steps involved in DIR24-8 based LPM6 matching algorithm is proportional to IPv6 prefix length and in each step, algorithm accesses a new TRIE node which is an expensive memory operation. In real world scenario, IPv6 prefixes can be stored in compressed form in TRIEs either because they are unique or because they have common prefixes. In compression algorithm, we have merged a child node with its parent if it is the only child of its parent.

Experiment: We have populated 20,000 IPv6 prefixes, all having length 48, with varying degree of compression possible.

Results: In the best case, when there were three lookups saving on an average, we have got 38% higher throughput over DIR24-8 based TRIE and in the worst case, when there is no compression possible, compressed TRIE showed only a dip of 3.3% in overall throughput. On an average, with 1.25 lookups saving, we got 16.1% higher throughput than DPDK LPM6 lookup algorithm.

4 RELATED WORK

CPU based packet processing: RouteBricks[8] is one of the first paper in this area. They exploited various components in a commodity server to achieve 35 Gbps throughput for Layer 3 Forwarding. They used both inter-server and intra-server optimizations to achieve this throughput.

Manual optimizations: There are many papers where authors have come up with different kind of manual optimizations to show the improvement in the performance. Batching[8, 10, 11, 15] has been used extensively by authors, however, these papers are determining the batch size by empirical analysis. [10, 15] are exploiting the fact that CPU and Memory subsystem can work in parallel and memory stall can be minimized by issuing the software prefetches before actually using the data. However, it is difficult to use these manual optimizations each time we are writing a packet processing application.

Compiler optimizations: Shangri-La [6] generates optimized binary for network processor and showing that the generated binary is working as good as hand-tuned code. [7] talks about the importance of doing the optimizations in the compiler rather than hand-tuning the same thing for different applications. The main focus of paper [7] is to automating the decision of breaking the application in parallel components to achieve high throughput. Due to space constraint, we are not mentioning papers related to various kind of software based packet processors and different DSLs for writing the network applications.

In our L2 forwarding example, there are two table lookups, both of which can potentially result in main-memory access (and CPU stalls). Indeed the CPU stalls only occur if the lookup table cannot fit in memory *and* the entry being looked-up is not already present in the cache. Both these criteria are impossible to judge statically, and an automatic compiler-based approach enables online dynamic code regeneration to adapt to the workload needs.

5 CONCLUSION

The goal of this paper is to find the efficient batch size and right prefetch distance to use the underlying hardware efficiently and to improve the application performance. In the evaluation section, we are showing that per core performance for different applications is on par with hand-tuned optimized applications and applications are scaling with the number of cores. It saves a lot of time and efforts, and we don't need to think about the code flow for different types of applications and the possible bugs due to manual intervention. We believe that DSLs won't be very useful if we are unable to develop good compilers. The actual power of DSLs can only be realized when the compilers can generate the optimized target which is on par with hand-tuned code.

REFERENCES

[1] *Intel Data Plane Development Kit.* http://dpdk.org/.
[2] *Intel Xeon Processor E5-2640 v3.* http://ark.intel.com/products/83359/Intel-Xeon-Processor-E5-2640-v3-20M-Cache-2_60-GHz.
[3] *L2 Forwarding with Crypto.* http://dpdk.org/doc/guides-16.07/sample_app_ug/l2_forward_crypto.html.
[4] *Pktgen-DPDK.* http://dpdk.org/browse/apps/pktgen-dpdk/refs/.
[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890
[6] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. 2005. Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* ACM, New York, NY, USA, 224–236. https://doi.org/10.1145/1065010.1065038
[7] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. 2010. Controlling Parallelism in a Multicore Software Router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '10).* ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/1921151.1921154
[8] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 15–28.
[9] Pankaj Gupta, Steven Lin, and Nick Mckeown. 1998. Routing Lookups in Hardware at Memory Access Speeds. 1240–1247.
[10] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. 2015. Raising the Bar for Using GPUs in Software Packet Processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* USENIX Association, Oakland, CA, 409–423. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kalia
[11] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. 2012. The Power of Batching in the Click Modular Router. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS '12).* ACM, New York, NY, USA, Article 14, 6 pages. https://doi.org/10.1145/2349896.2349910
[12] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
[13] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. 2016. High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16).* ACM, New York, NY, USA, 629–630. https://doi.org/10.1145/2934872.2959080

[14] Ting Zhang, Yi Wang, Tong Yang, Jianyuan Lu, and Bin Liu. 2013. NDNBench: A benchmark for Named Data Networking lookup. In *2013 IEEE Global Communications Conference, GLOBECOM 2013, Atlanta, GA, USA, December 9-13, 2013*. IEEE, 2152–2157. https://doi.org/10.1109/GLOCOM.2013.6831393

[15] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2535372.2535379