# 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 [5]). 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 [14], 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 preliminary 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 , , ,* 7 pages.



, ,

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

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 repetitive. For example, the RouteBricks paper [8] 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[13], 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. [12] 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 [14], 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 on general-purpose hardware, using an 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 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 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.

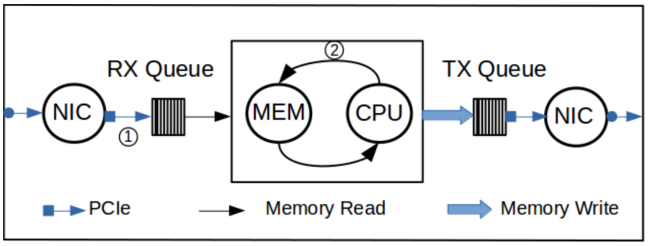


**Figure 1: Packet Processing Pipeline**

We first discuss the characteristics of the underlying hardware. Three components of a general-purpose architecture, are most critical to the performance of the packet-processing application: NIC subsystem, CPU, and Memory subsystem. Figure 1 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 DMA 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 DMA 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 allows 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 1.

The next step in each packet processing pipeline is the actual logic. 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 needs to 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 [11] 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. [11] 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 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 (② in Figure 1). 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, are loop-fission transformations [15], when viewed as a compiler optimization. Figures 2 and 3 show the batching and sub-batching transformations respectively. We use $B$ to denote the batch-size, and $b$ to denote the sub-batch-size.

```
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 2: Batching.**

```
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 3: Sub-batching, within `process_packet`.**

Sub-batching hides the CPU-memory latency. We further improve it by using the x86 *prefetch* instruction for future packets. The

[1]The total PCIe bandwidth to main memory in our experiments is around 8 GT/s, which gets shared across multiple NICs. For all our experiments, except one (IPv4), the total PCIe bandwidth remains unsaturated.

prefetch instruction allows the hardware to differentiate a memory request that is likely to be used in future, from a memory request that is likely to be used immediately (fetch), and allows better scheduling of resources by hardware. Algorithm 1 shows the example prefetching code used for the hash-lookup in the L2 forwarding example.

**Algorithm 1** HASH LOOKUP

```
1: for i ← 1 To BatchSize do
2:     key-hash[i] = extract key and compute hash;        ▷ C_K
3:     prefetch(bucket-for-key-hash(key-hash[i]));        ▷ C_P
4: end for
5:
6: for j ← 1 To BatchSize do
7:     value[j] = hash-lookup(key-hash[j]);               ▷ C_L
8: end for
```

Notice that sub-batching is dependent upon batching, in that, the sub-batch-size can only be smaller than the batch-size. Thus, if there is no batching, there can be no sub-batching. We find that the optimal sub-batch-size is often less than the optimal batch-size.

Finally, the processed packets are transmitted to the output NICs. Assuming uniform distribution of output packets across output NICs, we expect the utilization at the output memory-NIC DMA interface to be similar to the utilization of the input NIC-memory DMA interface.

We evaluate the effectiveness of the batching and sub-batching transformations, and systematically explore the solution space to search for the optimal values of $B$ (batch-size) and $b$ (sub-batch-size), in our experiments.

## 3 EVALUATION

We perform experiments to answer two main questions: (a) How much performance improvement is possible through carefully tuning the batch-size $B$ and the sub-batch-size $b$? (b) Can we develop a performance model from the experimental data, to generalize our results for automatic optimization by a compiler optimizer, for an arbitrary packet processing pipeline. We develop a preliminary model to explain our current results. We have integrated our model into P4C to automatically optimize the applications used in this paper.

### 3.1 Evaluation Setup

**Hardware**: 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**: 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.

**Methodology**: We use packet-processing pipelines developed in P4[5] as our test applications. We have extended the P4C[14] compiler to generate code with batching, sub-batching and prefetching. P4C[14] generates C-code that can be run with the DPDK[1] framework, which is then compiled into an executable using GCC. Unless otherwise specified, all applications are tested with minimum-sized packets (64 bytes), to test the limits of the system.

**Applications**: Our applications are similar to the ones used in [11], to allow head-to-head comparison of performance results:

(1) **Layer 2 Switch**: Two hash tables are used to store the mapping between SMAC/DMAC and the forwarding port. Each packet goes through two lookup stages, one for SMAC and another for DMAC. By default, we assume 16 million entries in both tables (as also used in [11]), unless otherwise specified.
(2) **IPV4 Forwarding**: A longest-prefix match (LPM) lookup is performed on the destination IP address to get the forwarding port. We populate the forwarding table with 527,961 prefixed, as also used by [11].
(3) **IPv6 Forwarding**: A longest-prefix match lookup is performed on the destination address to find the egress port. We populate the DPDK LPM table with 200,000 random entries with the length between 48 to 64, as also done in [11]. Through Pktgen, we generate packets with destination address randomly picked from these 200,000 entries. The minimum packet size for this application is 78 bytes and not 64 bytes.
(4) **Named Data networking**: A hashtable lookup is used to map a string URL to a forwarding port. We use the URL dataset given in [16]. Using Pktgen, we transmit packets containing URLs generated randomly from our dataset. We use 32 bytes URLs in our packet headers, as also done in [11].
(5) **L2 forwarding with encryption and decryption**: We use the L2Fwd-Crypto[3] application available as a part of the DPDK source code. The application performs encryption and decryption based on the input parameters and it then forwards the packet on Layer 2 with static port mapping. Unlike other applications, which are largely memory-bound, this application is compute-bound.

### 3.2 Performance improvements achievable through batching and sub-batching

We added batching and sub-batching support to P4C, and show results for batch-size $B = 32$ in Figure 4. The `Batching` results represent the case when sub-batching is disabled (i.e., sub-batch-size $b = 1$), while the `Batching and Prefetching` results represent the case when sub-batching (and prefetching) is enabled and set to its maximum possible value, i.e., sub-batch-size $b = 32$. We later

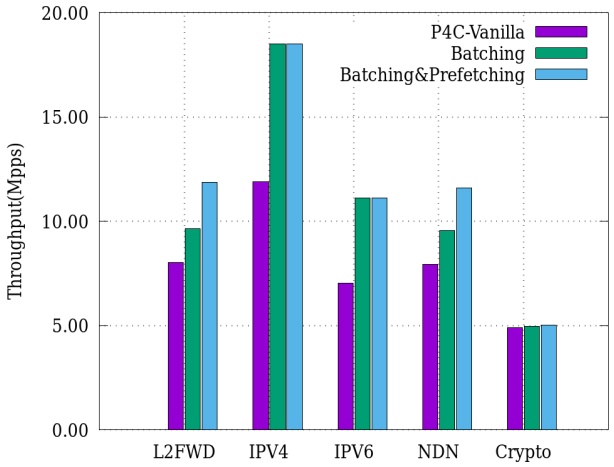


**Figure 4: Effect of batching and prefetching**

discuss the sensitivity of throughput to changing $B$ and $b$. For L2 forwarding, batching improves the performance by 20% and sub-batching further improves the performance by an additional 23%. Similarly for NDN, batching alone improves the performance by 20% and sub-batching results in an additional performance gain of 21%. Using $B = 32$ and $b = 32$ for IPv4 and IPv6, we obtain a performance gain of 55% & 57% respectively. For a compute-intensive application like L2Fwd-Crypto, the performance improvements are much smaller.

## 3.3 Sensitivity of Throughput to $B$

We use the L2 forwarding application to demonstrate the effects of changing application behavior on the optimal values for $B$ and $b$. We use the L2Fwd application with five different sizes of the lookup table. The size of the lookup table approximately represents the working set of the application. If the table fits in the caches, then the application is largely compute bound and has few accesses to the main memory. On the other hand, if the table is much larger than the last-level cache, then almost every random table access will result in an access to the main memory. Consequently, the changing application behavior results in a change in the value of the optimal $B$ and $b$, required for optimal throughput.

Figure 5 plots the results for throughput for different table sizes and batch-sizes $B$. The solid line represents the case when the sub-batch-size $b$ equals the batch-size $B$, i.e., $b = B$. The dotted line represents the case when the sub-batch-size $b = 1$, i.e., no sub-batching.

There are a few conclusions to draw from this plot: (1) Expectedly, throughput is generally higher for smaller tables than for larger tables. (2) For all table sizes, the throughput usually increases with increasing $B$ for $B < 128$, due to better exploitation of DMA bandwidth. (3) A batch-size beyond 128, usually results in decreased throughput due to greater stress on the caching subssystem. (4) Sub-batching improves the throughput for larger table-sizes, but does not improve throughput for smaller table sizes. This is expected because sub-batching benefits from memory-level parallelism. For small table-sizes, the main-memory accesses are few, and so the benefit of sub-batching is little. We discuss this in more detail in our next experiment. (5) $B = 32$ provides near-optimal results across all configurations of the L2 forwarding application.

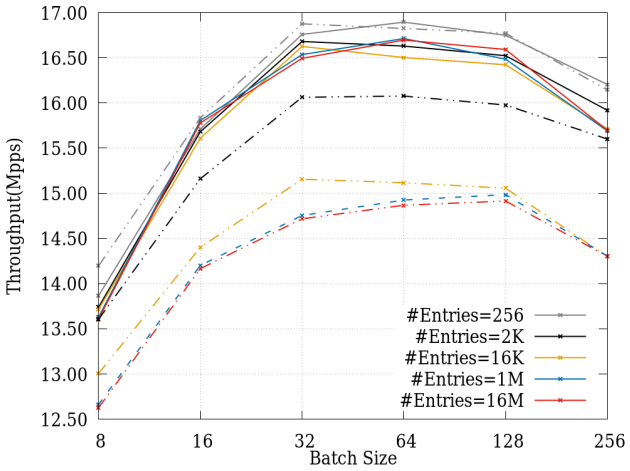


**Figure 5: The effect of batch-size $B$ on application throughput, for five different configurations of the L2 forwarding application. The solid line represents $b = B$ (sub-batching enabled), and the dotted line represents $b = 1$ (sub-batching disabled).**

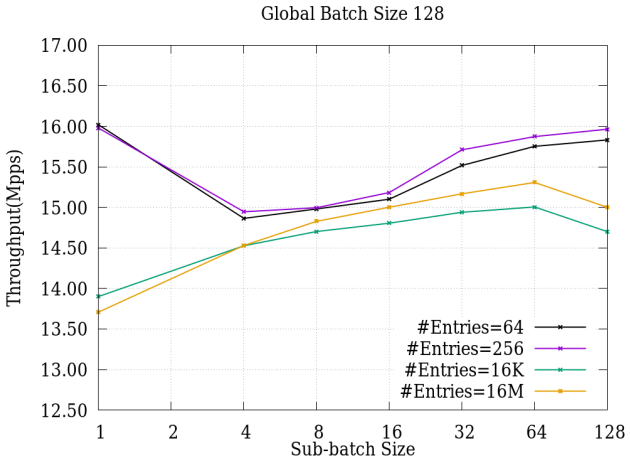


**Figure 6: The effect of sub-batch-size $b$ on application throughput, for four different configurations of the L2 forwarding application. We use batch-size $B = 32$ for these experiments.**

## 3.4 Sensitivity of Throughput to $b$

To further understand the sensitivity of application throughput to the sub-batch-size $b$, we vary $b$ for a fixed value of $B = 128$, for the

L2Fwd application. Figure 6 plots the results, again for different table-sizes. It is interesting to see that if the table-size is less than, or equal to 256 entries, the throughput *decreases* with increasing batch-size. On the other hand, if the table-size is large (indicating that many table accesses will be cache misses), the increasing sub-batch-size, generally improves throughput. If the sub-batch-size is too large, (e.g., if it is 128 in this case), the performance usually dips due to increased pressure on the caching subsystem, caused by the increased memory footprint.

These results confirm that while sub-batching is useful for memory-bound applications, it is not effective (or sometimes even harmful) for compute-bound applications. We try and capture this formally in our performance model, discussed next.

## 3.5 Performance model

We discuss a performance model to explain our experimental results. At a high-level, the system can be modeled as a queueing network, with three parallel relevant server components: namely, the CPU, the CPU-memory interface, and the I/O-memory DMA interface. Assuming that the three components can execute in parallel, and have service rates $c$, $m$, and $d$, respectively, the final throughput of the system would be:

$$min(c, m, d)$$

In other words, the throughput of the system is limited by the slowest component (each packet is expected to traverse all three components).

With batching, we allow multiple in-flight packets on the DMA interface, and hence increase the parallelism and service rate $d$. If the batch-size is $B$, then the ideal improvement in the service rate of the DMA interface would be $B * d$ (i.e., linear scaling). However, in practice, the interface is not fully parallelizable, and so the service rate of the DMA interface increases as some function $f_{dma}(d, B)$. The plot in Figure 5 can be used to roughly approximate the shape of this function $f_{dma}$ with increasing value of $B$.

Similarly, with sub-batching, we allow multiple in-flight memory requests on the CPU-memory interface, and hence increase the service rate $m$. If the batch-size is $b$, assume that the service rate of the CPU-memory interface is denoted by a function $f_{mem}(m, b)$. The shape of $f_{mem}$ can be roughly approximated using the plots in Figure 6. As we can see, the shape of both functions $f_{dma}$ and $f_{mem}$ are dependent on the characteristics of the application. For example, if the application is compute-bound, $f_{mem}$ typically decreases with increasing $b$. On the other hand, $f_{mem}$ increases with increasing $b$, for memory-bound applications, if $b < 128$.

Finally, if any improvements can be made to the CPU processing logic (e.g., see discussion in Section 4), then those are counted towards improvements in $c$. In summary, with batching and sub-batching, the throughput of an application can be represented as:

$$min(c, f_{dma}(m, b), f_{mem}(d, B))$$

Thus, a compiler needs to first estimate the characteristics of the application. For example, the compiler may run the application for different configurations in an offline phase, to make conclusions about the degree of compute-boundedness ($c$), memory-boundedness ($m$), and I/O-boundedness ($d$) of that application. Then, given a machine, the compiler may use this information with the model presented above to decide the optimal values for $b$ and $B$.

While our model is preliminary, it does provide an initial basis for reasoning about performance while generating code for packet processing pipelines.

## 3.6 Comparison with other related work

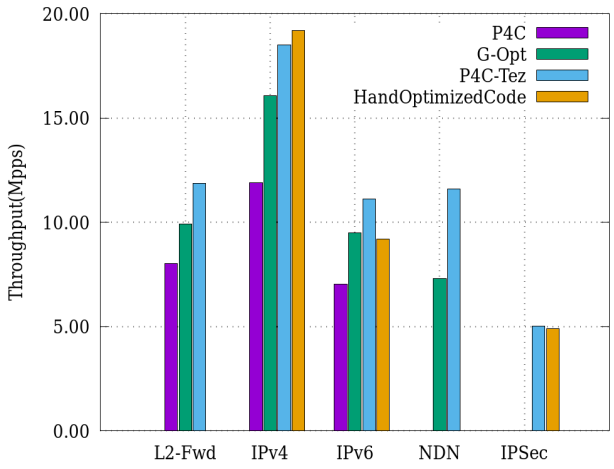


**Figure 7: Comparison with other related work.**

Figure 7 compares our automatic generated code with vanilla P4C [14], with another related work, G-Opt, aimed at extracting memory-level parallelism through manual annotations [11], and comparable hand-optimized code available as a part of the DPDK distribution. To ensure fair comparisons, we verify that the number of lookups and number of entries in the table(s) is same in all experiments. In cases where a hand-optimized version or G-Opt results are not available, the corresponding throughput bar is omitted.

Comparing with vanilla P4C, we obtain 48%, 55%, 57%, and 46% throughput improvements for L2Fwd, IPv4, IPv6, and NDN applications respectively. The L2Fwd-crypto application shows minimum improvements, due to its compute-intensive nature.

G-Opt is a previous effort at bridging the gap between CPU and GPU performance for packet processing applications. The G-Opt authors manually annotate code to identify expensive memory accesses, and use multi-threading and context-switching to hide the memory latency. There are two important ways in which our work differs from G-Opt: (1) our approach is largely automatic and works with a high-level program representation. (2) we employ batching and sub-batching instead of multi-threading, which makes our approach more efficient as it avoids context-switching overheads. We report a 20%, 15%, 17%, and 59% improvement over G-Opt for L2Fwd, IPv4, IPv6, and NDN applications respectively, in head-to-head comparisons. We find that G-Opt authors did not systematically explore the space of all possible transformations, when compared with our work, perhaps due to the lack of a performance model. Our systematic exploration of $B$ and $b$, allows us to maximize the throughput, beyond previous work.

Comparing with hand-optimized code available as a part of the DPDK distribution, we find that we are sometimes slightly worse (4% worse for IPv4), and sometimes significantly better (20% better for IPv6). It is not surprising that our compiler-based approach of systematic exploration across the parameter space, can perform better than hand-optimized code written by developers.

### 3.7 Scalability

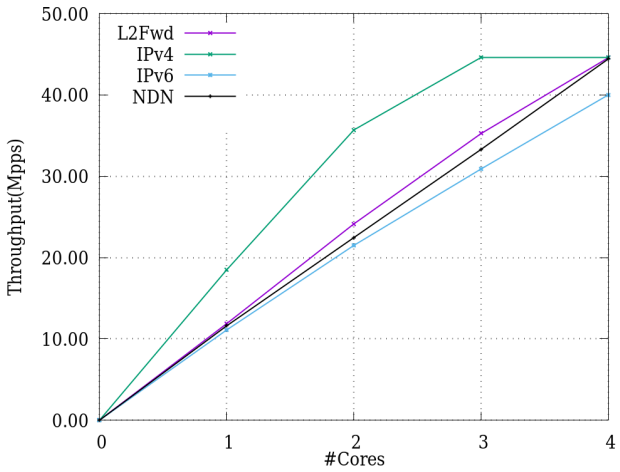


**Figure 8: Number of Cores vs Throughput**

Figure 8 plots the application throughput with increasing number of cores. Unsurprisingly, the throughput usually scales linearly with the number of cores, except when they saturate the PCIe bandwidth. In our experimental setup, the theoretical max achievable throughput for 64 byte packets, summed across 4 NICs, is 59 Mpps (million packets per second). However, our experimental results show that it cannot sustain this theoretical limit, and instead saturates at 44Mpps. This difference between theoretical and observed throughputs, is attributed to the fact that the same NICs are being used to both send and receive packets, resulting in complex scheduling dependencies at the PCIe level. IPv4 throughput hits the PCIe bandwidth limit in our experimental setup (the hash lookup algorithm for IPv4 is the fastest among all applications). IPv6 packet throughput is slightly lower than others, because it uses larger (78-byte) packets.

### 3.8 Latency

In all our experiments, the end-to-end latency of a packet can be at most $B$ times worse than optimal. Usually, trading-off increased latency (in an already fast end-to-end system) for better throughput, is acceptable for most real-world applications.

## 4 DISCUSSION

While we restricted our discussion to batching and sub-batching, compiler transformations can be much more sophisticated. For example, the choice of data-structure to maintain and lookup the table, is often critical to the CPU speed of the application. We believe that existing work in the realm of data-representation synthesis in programming languages [9, 10], can be leveraged effectively in this setting. We discuss one such experiment where we optimized the trie data structure used for IPv6 longest prefix match, to reduce the number of average memory accesses per packet.

We implemented longest-prefix match lookup based on a compressed trie, and experimented with our IPv6 application, albeit with 20,000 IPv6 prefixes, all having length 48. We save around 1.25 memory accesses per packet for this application, resulting in 16% higher throughput. In general, the required transformations are highly dependent on the application (e.g., L2 forwarding vs. IPv6 lookup), and the data (e.g., table size), and these characteristics could change over time. We think that this argues for an automatic adaptive optimization engine, like the one we have prototyped inside P4C, to be able to efficiently utilize the available hardware resources.

## 5 RELATED WORK

There has been a plethora of related work on CPU-based packet processing and related programming models, including Click [13] and RouteBricks [8]. Manual optimizations to improve the performance of packet processing algorithms [8, 11, 12, 17] have also been studied extensively. In contrast, our work takes a compiler-centric approach to this problem, targeting micro-architectural optimizations, by systematically exploring the space of potential configurations. Our results surprisingly show performance improvements over these previous papers. While our approach is aimed at automatic code generation from a high-level specification, most previous works involved manual optimizations for individual applications. We have already compared our work with some of the most relevant previous work through our experiments.

Previous work in compiler optimization has close parallels with our work too. Shangri-La et. al. [6] generate an optimized binary for a specialized *network processor*, and show that the generated binary works as well as hand-tuned code. Dobrescu et. al. [7] automate the decision of splitting an application into parallel components to achieve high throughput. Previous work on data-representation synthesis [9, 10] involves automatically inferring the optimal data structures and schedule for a high-level program specification. Unlike this previous work, our focus on network processing pipelines on general-purpose hardware, enables more domain-specific optimizations, and deeper micro-architectural modeling for better overall performance.

While our initial efforts are aimed at optimization and performance modeling, we hope to extend this work, towards developing an adaptive optimizing compiler for packet-processing pipelines.

## 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] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1993498.1993504

[10] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent Data Representation Synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 417–428. https://doi.org/10.1145/2254064.2254114

[11] 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

[12] 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

[13] 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.

[14] 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

[15] M.J. Wolfe. 1982. *Optimizing supercompilers for supercomputers*. Ph.D. Dissertation. Univ. of Illinois,Urbana, IL.

[16] 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

[17] 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