Automated SmartNIC Offloading Insights for Network Functions

Yiming Qiu
Rice University
Houston, TX, USA

Jiarong Xing
Rice University
Houston, TX, USA

Kuo-Feng Hsu
Rice University
Houston, TX, USA

Qiao Kang
Rice University
Houston, TX, USA

Ming Liu
UW-Madison/VMware
Madison, WI, USA

Srinivas Narayana
Rutgers University
New Brunswick, NJ, USA

Ang Chen
Rice University
Houston, TX, USA

Abstract

The gap between CPU and networking speeds has motivated the development of SmartNICs for NF (network functions) offloading. However, offloading performance is predicated upon intricate knowledge about SmartNIC hardware and careful hand-tuning of the ported programs. Today, developers cannot easily reason about the offloading performance or the effectiveness of different porting strategies without resorting to a trial-and-error approach.

Clara is an automated tool that improves the productivity of this workflow by generating *offloading insights*. Our tool can a) analyze a legacy NF in its *unported* form, predicting its performance characteristics on a SmartNIC (e.g., compute vs. memory intensity); and b) explore and suggest *porting strategies* for the given NF to achieve higher performance. To achieve these goals, Clara uses program analysis techniques to extract NF features, and combines them with machine learning techniques to handle opaque SmartNIC details. Our evaluation using Click NF programs on a Netronome SmartNIC shows that Clara achieves high accuracy in its analysis, and that its suggested porting strategies lead to significant performance improvements.

CCS Concepts: • **Computing methodologies → Machine learning**; • **Networks → Network performance modeling**.

Keywords: Network function, SmartNIC, Machine learning

1 Introduction

High-speed networks are the backbone of datacenters, and their data rates are consistently increasing over the years [17, 30]. As a result, server CPUs spend more and more cycles on packet processing, and the consumed resources are no longer available to revenue-generating tenant VMs. The gap between CPU and networking speeds will further widen in the post-Moore era [28, 36]. This has motivated the development of SmartNICs [3, 8, 9, 11] for near-network processing.

Unlike traditional NICs with hardwired offloading modules (e.g., TSO/LRO/checksum), SmartNICs have programmable SoC cores, specialized packet IO engines, and various domain-specific accelerators. Researchers have leveraged SmartNICs as an offloading platform, and developed a range of network applications [42, 45, 47, 58, 59]. SmartNIC offloading enables server CPUs to perform more application-level work. Moreover, SmartNIC SoC cores are also more energy-efficient, driving down the total cost of ownership (TCO).

However, as a computing platform, SoC SmartNICs represent a significant departure from the familiar programming model and performance characteristics of x86 servers. This is especially true for "baremetal" designs [11–13, 33, 46] that are specifically optimized for packet processing tasks. For instance, these platforms are shipped with a lightweight runtime/firmware without OS or full `libc` support, they may customize their ISAs for packet operations, and they may expose a much constrained programming model (e.g., a subset of C). Netronome SmartNICs are a popular hardware of this kind [11, 52]. Emerging platforms like the Fungible data processing units [12] and Pensando distributed service cards [13] also share many similarities. On such platforms, NF performance is predicated upon the intricate interplay across many factors: the compute versus memory profiles of NF programs, the use of accelerators, hardware architectures and compiler optimizations, as well as the target workloads. This performance opacity has led to a cumbersome development process for NF offloading.

In order to understand and improve offloading performance, today's developers perform manual analyses and tuning [47, 58]. At the heart of this workflow is a *cross-platform porting* process. The developer needs to first rewrite the x86 code against SmartNIC toolchains, which are often closed-source and vendor-specific (e.g., the compiler and runtime). Further, she needs to perform hardware benchmarks on a representative workload, and re-optimize the ported program iteratively. If her assumptions about the workloads, architectural details, compiler optimizations, or the NFs are amiss, multiple rounds of hand-tuning would be required. Researchers and developers would significantly benefit from an

SOSP '21, October 26–28, 2021, Virtual Event, Germany
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483583

automated workflow for analyzing ported NF performance and identifying effective performance tuning strategies. In this paper, we call such information *offloading insights*.

Clara is an automated tool that generates offloading insights for SmartNIC NFs. As the first step, Clara directly analyzes an unported NF program, and *predicts* its key performance parameters—the numbers of compute instructions and memory accesses—when offloaded to a SmartNIC. Using such information as a starting point, Clara then automatically explores a plethora of possible optimizations for offloading. It suggests *effective porting strategies* to improve performance. This includes opportunities for using accelerators, multicore scaleout analysis, NF state placement strategies, memory access optimizations, and NF colocation analysis.

Clara achieves these goals by augmenting program analysis techniques with machine learning (ML) prediction. For performance estimation, Clara transforms the NF programs to a uniform LLVM IR (intermediate representation), and analyzes them with ML to infer how closed-source SmartNIC compilers might perform instruction selection and optimizations on the IR. To suggest porting strategies, Clara performs a training phase that searches through the program tuning space to understand the performance effects. It then uses the learned cost model to suggest NF- and workload-specific optimizations despite opacity in the SmartNIC compiler, hardware, and runtime details. As a combined effect, Clara relieves NF developers from the burden of having to reason through layers of complexity for performance optimizations. Equipped with Clara's offloading insights, developers only need to focus on getting the ported code functionally correct. We make the following contributions:

- Clara is the first tool to generate automated offloading insights for network functions on SmartNICs.
- It develops novel techniques to analyze unported NF programs, predict cross-platform performance parameters, and suggest effective program porting strategies to improve performance.
- We prototype Clara (code available at [4]) and apply it to Click NFs on a Netronome SmartNIC, generating accurate and effective offloading insights.

2 Motivation

Reasoning about network function performance is already a challenging task even for traditional x86 platforms [41, 49]. SmartNIC-offloaded NFs present further barriers—we need to reason about *cross-platform* performance characteristics, and different *porting strategies* usually result in different performance gains. A myriad of factors come into play.

Proprietary architectures and toolchains. SmartNIC architectures are significantly different from the familiar x86 platform, and they are further managed by vendor-specific toolchains; both are proprietary in nature [3, 8, 9, 11, 43]. Therefore, an accurate understanding of hardware details or

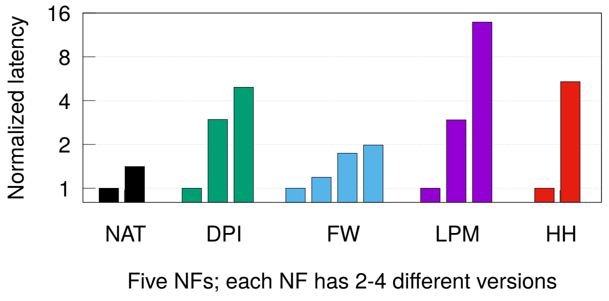
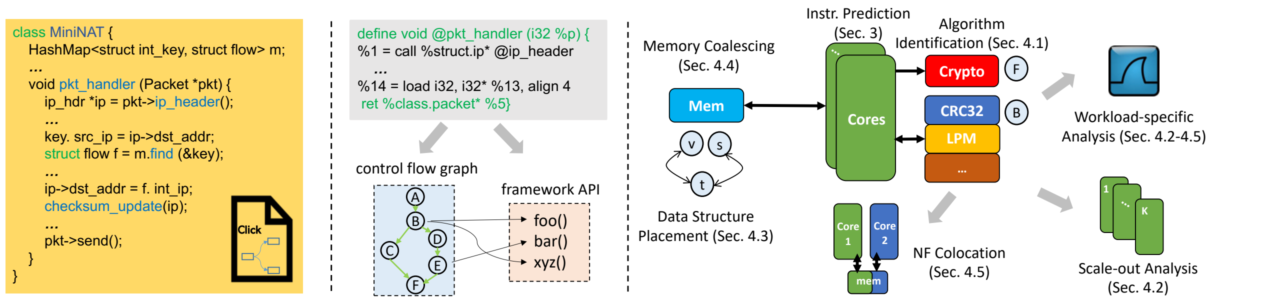


Figure 1. Performance variability of five network functions on a Netronome SmartNIC. For each NF, we benchmark two to four different versions with the same core logic. Network address translation (NAT) variants use checksum accelerators optionally. Deep packet inspection (DPI) variants handle different packet sizes. Firewall (FW) variants store flow state in different memory locations and have varying flow distributions. Longest prefix match (LPM) has different numbers of match/action rules and optionally uses the flow cache. Heavy hitter detection (HH) has varying packet rates. All NF latency are normalized against the fastest version.

the compiler internals is hard to come by. SmartNIC developers are limited by what they can gather from hardware "databooks". These documents showcase SmartNIC features in brief but do not contain full details about the design, a vague guide at best. In addition, the proprietary compiler may perform instruction selection or optimizations based on cost models and hardware details unavailable to developers.

Cross-platform porting and tuning. The developer also lacks access to reliable porting guidelines to improve offloading performance. On x86 platforms, developers can fine-tune their programs, even at assembly level, based on community-accrued experiences and best practices. In contrast, SmartNIC developers face a higher barrier due to the paucity of community expertise. They have to resort to a trial-and-error approach to experiment with different porting strategies. Program tuning also needs to account for the complex memory hierarchies, specialized ISAs and accelerators, and other types of architectural heterogeneity of SmartNICs. Moreover, the same program tuning strategy will have different effects depending on the NF programs and workloads.

Consider some concrete examples on the Netronome Agilio SmartNIC. The latency of LPM (longest prefix match) functions could vary by orders of magnitude depending on whether the program uses the "flow cache"—an accelerated mechanism for flow matches. Implementations that use the flow cache significantly outperform those that use regular match processing for cache hits. Header checksums require 2000+ cycles on the general-purpose cores, but only 300 cycles on ingress accelerators. Offloading performance also depends on the traffic profile, such as the number of concurrent flows, packet types, and packet sizes. For instance,



```
class MiniNAT {
    HashMap<struct int_key, struct flow> m;
    ...
    void pkt_handler (Packet *pkt) {
        ip_hdr *ip = pkt->ip_header();
        ...
        key. src_ip = ip->dst_addr;
        struct flow f = m.find (&key);
        ...
        ip->dst_addr = f. int_ip;
        checksum_update(ip);
        ...
        pkt->send();
    }
}
```

```
define void @pkt_handler (i32 %p) {
%1 = call %struct.ip* @ip_header
...
%14 = load i32, i32* %13, align 4
 ret %class.packet* %5}
```

Click
control flow graph
framework API
A
B
C
D
E
F
foo()
bar()
xyz()
Memory Coalescing (Sec. 4.4)
Instr. Prediction (Sec. 3)
Algorithm Identification (Sec. 4.1)
Mem
Cores
Crypto
F
CRC32
B
LPM
...
v
s
t
Data Structure Placement (Sec. 4.3)
Core 1
Core 2
mem
NF Colocation (Sec. 4.5)
Workload-specific Analysis (Sec. 4.2-4.5)
1
K
Scale-out Analysis (Sec. 4.2)

(a) Legacy NF (e.g., Click programs) (b) LLVM program preparation (c) Clara generates several types of offloading insights

Figure 2. The roadmap of Clara and the key techniques. (a) Clara takes in a legacy NF in its unported form (e.g., Click programs). (b) Clara transforms the NF into a low-level Intermediate Representation (IR) using program analysis. (c) Clara uses a combination of program analysis and machine learning techniques to generate several types of offloading insights.

TCP SYN packets may require flow state setup, and checksum or DPI operations would take different amounts of time depending on the packet sizes. Figure 1 further showcases a set of benchmarks for five NFs on Netronome Agilio. We benchmark each NF with 2–4 different porting strategies or workloads. Depending on the use of accelerators, memory locations, flow table sizes, and traffic profiles, the performance can vary up to 13.8×.

As we can see, the cross-platform porting process can be full of surprises if developers' assumptions about hardware, workloads, or NF programs are slightly off. Automated support for generating offloading insights will significantly benefit today's SmartNIC developers.

Clara: Automated offloading insights. Clara is a tool that automates much of this work by generating *offloading insights* for cross-porting network functions to SmartNICs. As shown in Figure 2, Clara analyzes an *unported* NF statically, even if it cannot directly run on the target SmartNIC, to predict its key performance parameters once cross-ported. Concretely, Clara estimates the number of compute instructions and the number of memory accesses, which are critical parameters for NF performance analysis [41, 49]. Furthermore, Clara suggests effective porting strategies to improve offloading performance in an NF- and workload-specific manner. Achieving these goals requires tackling two technical challenges. First, Clara needs to perform *predictive analysis* of a host-based NF without having to cross-port the code. Second, it needs to *explore the program tuning space* to identify effective porting strategies despite hardware and compiler opacity. Clara's roadmap is to use program analysis techniques as a starting point, transforming NF programs in a uniform low-level IR to extract code features. It then relies on machine learning techniques to infer blackbox compiler behaviors and construct accurate cost models of the underlying hardware. Clara enables a variety of use cases.

- *Cross-platform performance prediction.* Existing tools for NF performance analysis characterize the numbers of compute instructions and memory accesses, but they do so for already-working x86 programs [41]. Clara enables similar analyses across platforms, without requiring the NF programs to be ported first.

Section 3 details this step. Beyond this characterization, Clara explores and suggests a range of porting strategies to increase performance, which we elaborate on in Section 4.

- *Identifying accelerator usage.* Significant performance gains are possible with SmartNIC ASIC accelerators (e.g., LPM, CRC). Clara extracts program features from the input NFs and identifies the corresponding algorithms that can benefit from hardware acceleration.
- *Multicore scale-out analysis.* SmartNICs are multicore platforms that parallelize packet processing. Assigning more cores to an NF may increase throughput, but only up to a certain point, because packet processing can be bottlenecked elsewhere on the NIC architecture (e.g., memory subsystems). Clara can suggest close-to-optimal core counts for different NFs.
- *NF state placement.* NFs contain stateful data structures of varying sizes (e.g., flow tables, ACLs). Clara exploits the SmartNIC memory hierarchy to identify effective state placements to increase performance.
- *Memory access coalescing.* To further reduce memory access latency, Clara analyzes the access patterns of stateful variables and suggests variable allocation and packing strategies, as well as memory coalescing sizes.
- *NF colocation analysis.* Packet processing often requires the use of multiple NFs. Some NFs may be friendly for colocation on the NIC, but others may result in high contention. Clara performs pairwise ranking of colocation strategies to minimize performance interference.

Clara significantly reduces manual labor for NF developers, and its offloading insights lead to greater performance. We showcase the power of Clara by applying it to Netronome, a popular baremetal NIC [24, 34, 52] with specialized non-coherent RISC cores, a lightweight runtime, programmable in restricted C dialects (Micro-C). Emerging platforms like the Fungible DPUs [12] and the Pensando DSCs [13] share

```
function PredictOffloadingPerf(prog, nic)
    //Program preparation
    llir ← LlvmBytecode(prog)
    cfg ← GetCfg(llir); api_set ← GetApi(llir);
    nf_blocks = GetCodeBlock(cfg);
    //Approximate compilation via LSTM
    for block ∈ nf_blocks do
        llvm_seq ← LlvmSeq(block)
        instr_seq ← LSTM(llvm_seq, nic)
        Insights.add(type=compute, block, instr_seq.compute)
        Insights.add(type=memory, block, instr_seq.memory)
    //Framework API analysis via reverse porting
    for api ∈ api_set do
        api_prof ← ReversePortPerf(api, nic)
        Insights.add(type=api, api, api_perf)
return Insights
```

Figure 3. The algorithm for approximate analysis of unported NFs to predict its ported performance parameters.

similar characteristics, all representing a drastic departure from general-purpose x86 platforms.

3 Predicting Offloading Performance

The first class of offloading insights center around predicting NF performance parameters on the SmartNIC. To obtain such information, Clara performs predictive analysis by extracting NF features using program analysis and inferring compiler behaviors via learning. First, Clara converts a legacy NF to a uniform LLVM [44] Intermediate Representation (IR), and constructs its control flow graph (CFG). Clara then predicts what the compiler might generate, at the instruction level, without knowing the exact hardware details, instruction semantics, cost, or compiler optimizations. This is achieved by training a machine learning model that *mimics compiler behaviors*. Clara also handles host and SmartNIC NF framework differences (e.g., Click API calls vs. Netronome built-in libraries) by a reverse porting process to achieve high fidelity in its analysis. Figure 3 summarizes the algorithm.

3.1 Program preparation

Clara first transforms an input NF program into a uniform format in LLVM intermediate representation (IR). The LLVM IR is in static single assignment (SSA) form, a simpler and more unified syntax that is easier to analyze compared to higher-level languages like C/C++. To ensure that the IR stays as close to the original NF logic as possible, Clara disables most LLVM optimizations when generating the IR representation of the NF program. Clara also extracts the control flow graph (CFG) of the NF program: nodes are basic code blocks without branches or loops, and edges represent branching or loop conditions.

```
class MiniNAT {
   HashMap<struct int_key, struct flow> int_map;
   void simple_action(Packet *pkt) {
      // if packet comes in from internal port
      ip_hdr *ip = pkt->ip_header();
      tcp_hdr *tcp = pkt->tcp_header();
      u16 hdr_size = (ip->ip_hl + tcp->th_off) << 2;
      if (hdr_size < ip->ip_len) {
         struct int_key key;
         key. src_ip = ip->dst_addr;
         key. dst_ip = ip->src_addr;
         struct flow f = int_map.find(&key);
         if (f != NULL) {
            ip->dst_addr = f. int_ip;
            tcp->dst_port = f. int_port;
            pkt->send(config_port);
         } // else if flow does not exist
      }
   }
};
```

Figure 4. A simplified Click element (NAT).

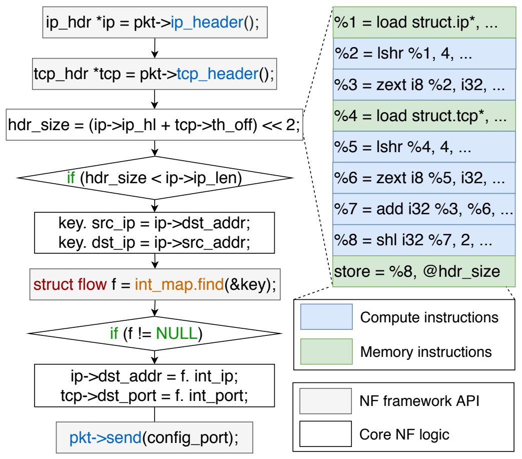

ip_hdr *ip = pkt->ip_header();
tcp_hdr *tcp = pkt->tcp_header();
hdr_size = (ip->ip_hl + tcp->th_off) << 2;
if (hdr_size < ip->ip_len)
key. src_ip = ip->dst_addr;
key. dst_ip = ip->src_addr;
struct flow f = int_map.find(&key);
if (f != NULL)
ip->dst_addr = f. int_ip;
tcp->dst_port = f. int_port;
pkt->send(config_port);
%1 = load struct.ip*, ...
%2 = lshr %1, 4, ...
%3 = zext i8 %2, i32, ...
%4 = load struct.tcp*, ...
%5 = lshr %4, 4, ...
%6 = zext i8 %5, i32, ...
%7 = add i32 %3, %6, ...
%8 = shl i32 %7, 2, ...
store = %8, @hdr_size
Compute instructions
Memory instructions
NF framework API
Core NF logic

Figure 5. The program preparation process.

Figure 4 shows an simplified Click element as an example of Clara input. The Click framework is a popular choice for NF prototyping, and many projects have chosen to base their designs upon this framework [26, 32, 49, 71]; Clara adopts the same assumption. Figure 5 further illustrates the Clara analysis on the NAT element. Clara annotates each LLVM instruction in a static pass, separating compute instructions (e.g., add), memory accesses (e.g., `load`), and NF framework API calls that need to be reverse ported (e.g., `call @ip_header`). It further distinguishes two different types of accesses: those to stateless NF variables, such as function-local stack variables that are temporary for each packet, and those to stateful NF variables that represent global data structures that persist across packets. Subroutines in the NF that do not depend on the host framework are directly inlined.

3.2 Predicting cross-platform performance

Clara then performs predictive analysis on the NF performance parameters on the SmartNIC, using the LLVM IR obtained from the step above. It further divides its analyses into two parts: one for stateful variable accesses (i.e., loads

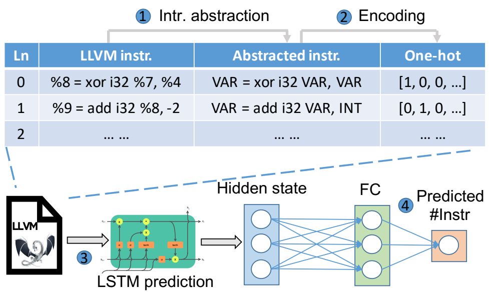


Figure 6. The instruction prediction machine learning model. The LSTM recurrently takes in LLVM instruction sequence encodings, and outputs a hidden state which represents the extracted information; The information is then fed into a Fully Connected (FC) layer for regression—i.e., predicting the number of instructions.

and stores to global variables in memory), and another for stateless computation (i.e., compute instructions, or accesses to function-local variables). As SmartNICs come with simpler microarchitectures (e.g., for memory reordering and prefetching), we find that stateful memory accesses have a clear correspondence to the load/store instructions at the IR level. Simply counting the number of memory instructions already leads to an accuracy of 96.4%–100% in practice. Stateless instructions, however, go through more layers of optimization: the compiler performs instruction selection or peephole optimizations to rewrite compute instructions; it also performs advanced register allocations for local variables so that stack operations may not result in any memory accesses. As these optimizations are dependent on the closed-source hardware execution model, constructing analytical techniques to infer such optimizations based on LLVM instructions is challenging. Instead, Clara *mimics compiler behaviors* via learning to analyze what might be produced for stateless instructions. By doing so, Clara estimates the numbers of compute instructions versus memory accesses, which are two key parameters determining NF performance [41].

Our choice of ML techniques is inspired by the recent successes of NLP (natural language processing) techniques, e.g., for sentiment analysis and text classification, for which analogies exist. Natural language processing is powered by sequence prediction—e.g., word mappings from source to target languages are based on best fits, and collections of words may appear together as phrases. In our problem, LLVM IR is the source language, the SmartNIC assembly is the target; each instruction is viewed as a word, and each code block as a sentence. Clara builds upon these analogies and applies neural language translation to predict the SmartNIC instructions from LLVM input. However, there are two unique challenges Clara addresses.

Data synthesis. NLP techniques require a large amount of document corpus from which patterns can be learned, but in our case SmartNIC programs do not exist in abundance due to the recency of such platforms. Therefore, preexisting LLVM/assembly program pairs are not easy to obtain. To address this, Clara borrows insight from *data synthesis* [25], a technique that the ML community uses to enhance training data by automatically synthesizing data pairs—which in our context are host NFs and their SmartNIC equivalents. Generating random yet representative programs is an active research topic on its own, and we build upon the progress made by the PL community in this regard.

Metric	Clara	Baseline
Jensen-Shannon divergence	0.0303	0.1010
Rényi divergence	0.1202	0.4061
Bhattacharyya distance	0.0354	0.1263
Cosine distance	0.0267	0.1164
Euclidean distance	0.0611	0.1383
Variational distance	0.3070	0.6713

Table 1. The Clara data synthesis engine generates representative Click programs. The metrics measure the distance between the instruction distributions for real-world vs. synthesized Click programs as compiled by LLVM.

Clara customizes YarpGen [48], a state-of-the-art tool that generates C/C++ programs by first randomly picking some ASTs (abstract syntax trees) and fleshing them out into source code. The AST generation strategy is further guided by the statistical properties of the target program corpus—e.g., Click elements have different syntax distribution from other programs. Our tool first analyzes existing Click elements to obtain representative AST distributions, and then feeds such properties to the program generator. Furthermore, Clara modifies YarpGen so that a) the emitted programs are encapsulated in C++ classes that inherit the 'public Element' class, and b) that the program statements manipulate packet headers, payload, and metadata as defined in the 'Packet' and 'WritablePacket' Click classes. As an additional constraint, Clara only targets packet operations that have corresponding support in the SmartNIC. Finally, the data pairs are then compiled by host and the SmartNIC compiler, respectively, resulting in LLVM instructions and SmartNIC assembly code. We have added and modified 1400 lines of code to YarpGen for this customization.

Table 1 shows that our synthesizer generates representative Click programs. We have compiled all synthesized programs using LLVM and collected the instruction distribution, and compare it against the distribution obtained from real-world Click programs. Using common similarity metrics that measure the quality of synthesized data [63], we found that our synthesizer achieves high fidelity. We also show a baseline synthesizer that does not take into account Click's AST distribution. The synthesized distribution is notably different from that of real-world Click programs.

Vocabulary compaction. A natural language has a well-defined vocabulary, but an instruction with the same opcode could be applied to an unbounded number of operands—e.g., different program variable names or constants could lead to an almost-infinite vocabulary. Clara compacts the vocabulary by abstracting away concrete variable names and substituting an operand with its type (e.g., 'add int const' instead of 'add x 2'), with the exception of well-defined header field names. This reduces the vocabulary to a few hundred distinct "words", representing a much smaller vocabulary than most natural languages. This compacted vocabulary enables Clara to apply a basic, *one-hot encoding* to the instructions, which essentially treats every word as a unique feature, while still achieving high accuracy. In contrast, NLP techniques often need to reduce the number of features by producing "word embedding" (i.e., associating words with their contexts), as the vocabulary is significantly larger.

With these enhancements, Clara trains a Long-short Term Memory (LSTM) [38], a recurrent neural network that is specialized for sequence prediction and used for language translation tasks. The output of LSTM is further fed into a set of fully connected (FC) layers that predicts the number of compute and memory instructions that the input program would compile to. During training, our loss function measures the differences between the predicted number and the ground truth. The training aims to learn the neural network weights that minimize such error. Once the training convergences, Clara uses the LSTM+FC model to perform inference to analyze a given NF program.

3.3 Handing NF framework API

The above analyses apply to LLVM instructions compiled from the NF body and its subroutines (by inlining the code), but NF framework API calls are handled separately—their implementations are different in nature across host and SmartNIC platforms. Clara must account for these framework differences to achieve a high-fidelity analysis.

As identified in a recent work, Gallium [71], the Click framework contains two classes of API calls: *stateless* header manipulation functions (e.g., 'ip_header', 'tcp_header'), and *stateful* data structures (most commonly, 'HashMap' and 'Vector'). The first class of API calls rely on Linux code to parse packet headers (e.g., 'struct sk_buff'), but SmartNIC versions would rely on their own packet handling functions (e.g., 'nbi_meta_pkt_info'). The second class includes stateful data structures. Click data structures (e.g., HashMap) elastically scale at runtime via entry insertions and deletions; but on baremetal NICs, data structure sizes must be pre-specified as they lack support for runtime memory allocation. The algorithms for walking the data structures are also different: a 'HashMap.find()' call in Click uses linear probing to resolve hash collisions, but the same API in Netronome uses a fixed set of buckets as dynamic memory allocation is prohibited. As another example, the 'Vector.delete()' Click call shrinks the data structure; but in Netronome, deletion calls only mark the entries as invalid. Instead of inlining code from the host framework, Clara substitute the framework API calls with what would faithfully represent the SmartNIC implementations, using an idea that we call *reverse porting*.

Reverse porting. We derive an additional set of Click elements that represent SmartNIC-style implementations of the same functions (e.g., fixed slots instead of linear probing). This is achieved by a *reverse* port of these SmartNIC libraries. We draw inspirations from the line of work in *transpilation* [2, 6, 20, 66]. Transpilers, unlike compilers that transform programs into a lower-level representation, translate from one source language to another. This is particularly useful for converting programs in arcane languages (e.g., Fortran) to easier languages (e.g., Python) where developer skills are readily available [7, 20]. We note on the analogy: SmartNIC programs are more restricted and arcane than C++. Transpilers follow a standard approach that applies manually-crafted substitution rules to program tokens (e.g., Fortran 'integer' is converted to 'int' in Python, 'logical' to 'bool', and 'dimension(0:n-1)' to 'array'); occasional errors are fixed manually to produce valid code [20]. We have followed a similar approach in Clara.

An important property this provides is the *symmetric* code structures in the reverse-ported code, as we simply take the AST (abstract syntax tree) of the source program and perform line-by-line translation following the same control flow. By ensuring that the newly derived Click elements follow the same control flow behaviors with the SmartNIC libraries, the same traffic workload will trigger similar processing behaviors in the reverse-ported code. This is especially critical for stateful data structures, because NF state maintenance requires complex branching and loops. Asymmetric code paths would lead to different execution and performance characteristics across platforms. To analyze the performance of the reverse-ported Click elements, Clara uses the machine code as compiled from the SmartNIC compiler directly instead of using learning-based inference.

4 Identifying Porting Strategies

Leveraging the numbers of compute instructions and memory accesses obtained above, Clara further suggests another class of offloading insights, which identify effective porting strategies in an NF- and workload-specific manner. The porting strategies cover accelerator usage, multicore scale-out factor analysis, NF state placement, memory access optimizations, and NF colocation suggestions.

4.1 Algorithm identification

Clara automatically suggests SmartNIC acceleration opportunities. Certain packet-processing algorithms (e.g., CRC checksum, longest prefix match) that are procedurally implemented in the host NF will benefit from specialized ASIC accelerators in the SmartNIC. Clara identifies such code blocks and suggests rewriting strategies.

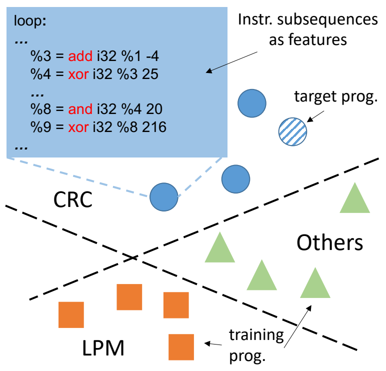


Figure 7. Clara extracts LLVM sequences as features, and classifies NF components to identify accelerator opportunities when porting the NF.

Recognizing opportunities for accelerator offloading, however, presents a technical challenge—the same functionality can be implemented differently by different developers, sometimes in idiosyncratic manners. Consider CRC (cyclic redundancy check) as an example, which computes a checksum over input data by padding, chunking, and looping through the chunks to perform XOR operations. A range of implementation strategies exist, and they differ in padding strategies, endianness considerations regarding bit and byte orders, choices in memorizing lookup tables—not to mention that CRC algorithms are further parameterized by the choice of CRC polynomials and checksum widths (e.g., CRC16 vs. CRC32). Similarly, a longest prefix match algorithm may use range or Patricia tries alike. Clara's analysis must consider the diversity of implementation choices for the same algorithm. Our observation is that each such algorithm has its inherent logical workflow, which exhibits distinct features under the ML lens despite implementation differences. As such, Clara uses learning to perform "pattern matches" against well-known accelerator algorithms in SmartNICs.

This is a classification task, where the learning process aims to identify classifiers—or separating curves in a feature hyperplane—that divide positive examples (e.g., CRC functions) from negative ones (i.e., non-CRC functions). In the inference phase, a new problem instance is compared against known instances for binary classification. Concretely, for the training process, Clara curates algorithm samples for common accelerators such as CRC hash functions and longest prefix matches. The dataset contains 600+ Click NF elements, and also 9000+ standalone programs crawled from online repositories, some of which implement these algorithms in different contexts (e.g., CRCs are common for error detection in non-NF programs as well). This is a one-time effort that does not need to be repeated unless there are new, uncaptured accelerator algorithms. The curated dataset is also incrementally expandable to incorporate new accelerators without relabeling the existing corpus.

We use support vector machines (SVM), which is a well-studied technique for ML classification [35]. It takes in training data points and their manually assigned labels, and learns classifiers based on program features. Code features are extracted using the Sequential Pattern Extraction (SPE) algorithm [29], where each feature is a subsequence of LLVM instructions and a feature vector is obtained for each data point. Feature extraction optimizes for the following goals: (1) an identifying feature should occur in many programs with accelerator usage opportunities—a property known as high *support*; (2) such a feature almost never appear in non-accelerator programs—known as high *confidence*. We also augment this with manually extracted features—for instance, LPM algorithms have distinct pointer chasing behaviors, moving from one address to a child address in a bounded loop. By identifying and combining multiple features for each accelerator algorithm, we achieve low false positive and negative rates. The classification phase assigns a label to a new data point based on its feature vector representation. In our case, the label is one of the accelerators, or simply 'none' if no such algorithm has been detected. Clara iterates through all known accelerators, and uses the trained classifiers to label a given NF's code block.

4.2 Multicore scale-out analysis

SmartNICs use multicore parallelism to improve packet processing performance. For instance, Netronome SmartNICs come with a large number of wimpy cores (e.g., 60× 1.2GHz cores), and packets are processed in run-to-completion mode. Increasing the core count for an NF generally leads to higher throughput, but beyond a certain point, the throughput would plateau due to contention at the memory subsystem. Such contention will also cause the packet processing latency to continue to increase with more cores. Moreover, depending on the level of memory/compute intensity, NF programs will bottleneck at different core counts.

Clara enables multicore scale-out factor analysis by predicting the optimal number of cores for an NF program and traffic workload. Its goal is to navigate the latency/throughput tradeoffs at different operating points, and to identify the "knees" of the latency curves [56]. To handle platform opacity, we take an ML-based approach as inspired by TVM [23], which trains cost models despite the vendor-specificity of accelerators (e.g., GPUs and TPUs), and infers effective optimizations for a given tensor program. By separating the 'algorithm' (i.e., program logic) from its 'schedule' (i.e., strategies of execution), TVM-like approaches search through the schedule space to identify effective optimizations.

Clara operates analogously. First, it synthesizes a set of training programs, which cover a wide range of arithmetic intensity (i.e., compute instructions vs. memory accesses to different regions) and program sizes. In the training phase, Clara tries different 'schedules' (i.e., the number of cores) for each program. In this automated pipeline, programs are synthesized and deployed to the SmartNIC for performance testing under different traffic workloads (e.g., flow size distributions, packet sizes). By observing the performance impacts with different core counts, Clara trains a regression model based upon GBDT [22] to predict the best core counts for different NFs under a given workload.

4.3 NF state placement

Stateful NFs contain global data structures (e.g., hashmaps or vectors that maintain cross-packet state), and packet processing requires flow- or IP-level state lookups and updates. Unlike x86 systems, where all state is maintained in DRAM and the cache subsystem is hardware-managed, SmartNIC platforms have a more complex hierarchy. Netronome SmartNICs, for instance, have cluster local scratch (CLS), cluster target memory (CTM), internal memory (IMEM), and external memory (EMEM), with increasing sizes and access latencies. Some caches are hardware-managed, but there are also software-managed scratches. Furthermore, without OS or full `libc` support, stateful NFs cannot easily rely on familiar `malloc` or `brk` calls to allocate memory dynamically. Similar restrictions apply to deallocation calls. An important offloading decision is where to allocate memory for each stateful data structure. Clara generates offloading insights on effective placement strategies.

Clara analyzes the data structure sizes in the NF programs in conjunction with a workload-specific profile (i.e., a pcap trace, similar as in host NF analysis projects [41]). To obtain access frequencies, Clara runs the Click NFs, augmented with reverse-ported elements with identical control flows, on the host machine with the specified workload. It then computes a placement strategy by formulating an ILP (integer linear programming) problem that minimizes the overall access latencies to all data structures. Consider an NF with k stateful data structures, where $s_i, i \in [0..k]$ denotes its size and f_i its access frequency. Clara places it to one of the t memory locations, where $j \in [0..t]$ represents the j-th level of memory hierarchy, L_j its access latency, and C_j its maximum capacity. A set of binary variables p_{ij} indicates whether or not the i-th data structure should be placed to j-th memory location. The objective function of the ILP is to minimize the overall access latency, or $\min(\sum_{i,j} L_j \times p_{ij} \times f_i)$, while satisfying the following constraints:

$$\forall i, \sum_{j=1}^{t} p_{ij} = 1; \quad \forall j, \sum_{i=1}^{k} p_{ij} \times d_i \leq C_j$$

where the first set of constraints specifies that each data structure must be placed at some memory location, and the second set specifies that the data structure sizes should not exceed the overall memory capacity.

4.4 Memory access coalescing

Another type of memory optimizations that Clara enables is memory access coalescing. This reduces the number of memory accesses to global variables by packing certain variables close together and by tuning access sizes. On host platforms, entire cache lines (typically 64-byte in size) are brought from or evicted to memory upon accesses. So for such systems, when allocating a set of variables (e.g., a C struct), memory-based optimizations include packing related variables in the same cache lines to reduce round trips to memory, as well as reduce false sharing by carefully determining allocation addresses. However, SmartNICs present more challenges. Accesses to a memory hierarchy may not occur at 64-byte sizes unless there is a similar cache subsystem, and managing variables in scratchpads is entirely up to software. Clara analyzes which variables are frequently accessed together, and it suggests variable allocation strategies to pack such fields close to each other. Further, Clara suggests memory access sizes based on the packing decisions, so that entire packs of variables are read and written together to minimize memory latency. As a concrete example, consider the following code snippet for TCP processing:

```
// index flow table
tcp->th_sport = sport; tcp->th_dport = dport;
...
// generate ack
if (tcp->th_ack == iss + 1 && state == 0) { .. }
```

Clara optimizes for such access patterns by suggesting that 'sport' and 'dport' be allocated adjacent to each other in the SmartNIC program. Similarly, it suggests packing 'iss' and 'state' variables together, which are used for generating TCP acknowledgment packets.

Specifically, Clara clusters global variables into different packs based on access frequencies collected from the host platforms. For a target network trace, Clara generates an *access vector* for each such variable v. Suppose there are k code blocks, Clara would compute an access vector of that size for v in the following manner. First, Clara counts the number of accesses to v from the i-th code block, $c_i, i \in [1..k]$. It then normalizes c_i by the total number of accesses to v, and computes $p_i = c_i / \sum_{i=1}^{k} c_i$. The resulting vector $[p_1, \cdots, p_k]$ would encode the access patterns of v. The Clara clustering algorithm suggests that variables with similar access vectors should be allocated adjacent to each other, and that they should be fetched using coalesced accesses. The sizes of the coalesced accesses are set to match the size of the variable pack. This is achieved by a traditional K-means algorithm.

4.5 NF colocation analysis

In many scenarios, developers need multiple NFs for packet processing. Care must be taken when colocating NFs to the same SmartNIC—if the colocated programs are not "friendly" to each other, they will experience significant performance degradation. A recent project [49] has shown that on x86 platforms, NF colocation can cause a variable amount of interference depending on the NFs. Clara generates offloading insights on effective colocation strategies when there are multiple candidate NFs to choose from.

Specifically, Clara performs *pairwise ranking* [22] to determine which NFs are friendly to each other. Ranking tasks are by nature different from traditional regression or classification. We rely on a state-of-the-art framework, XGBoost [22], and its LambdaMART ranking algorithm for this task. In the training phase, Clara randomly selects training NFs to be colocated on the SmartNIC. By default, each NF is given the same amount of SmartNIC resources, although this can be configured differently by the user. As NF interference primarily stems from contention at the memory subsystems [49], Clara extracts features including: a) arithmetic intensity of each NF, b) the number of compute instructions for each NF, and c) the ratio between colocated NFs' arithmetic intensities. By sampling many data pairs and minimizing the pairwise loss during training, Clara learns an ML model for ranking the friendliness for colocation. Concretely, this is measured by the performance degradation when NFs are colocated: the collective colocation throughputs normalized by their non-colocation peak throughputs. In the analysis phase, Clara iterates through all NF pairs under consideration. It extracts the features as stated above, and uses the ML model to predict how friendly the NF pair will be if they are colocated. Such ranking helps developers to make informed decisions in effective colocation strategies.

5 Evaluation

In this section, we present a comprehensive evaluation to measure the accuracy of Clara's analyses and the performance improvements after applying the Clara insights.

5.1 Prototype and setup

We have implemented a prototype of Clara using around 7400 lines of code in Python and C++, consisting of three components: a) a program analysis engine that relies on LLVM to generate LLVM IR and control flow graphs, as well as for program analysis; b) a machine learning engine that uses Scikit-learn and Tensorflow libraries for predictions and offloading insights; and c) a set of utility tools for supporting functions, such as data synthesis and constraint solving.

Setup. All our experiments have been conducted on a Ubuntu 18.04 server with six Intel Xeon E5-2643 Quad-core 3.40 GHz CPUs, 128 GB RAM, and 1TB hard disk, which is equipped with a 40Gbps Netronome Agilio CX SmartNIC. The traffic workloads are generated from `trafgen` [16], a multithreaded benchmarking tool for packet generation. The timing measurements are obtained via the ingress and egress timestamps on the SmartNIC platform. To evaluate Clara, we have used both synthesized NF programs from our toolchain, as well as a set of real-world Click NFs as summarized in Table 2. These Click programs are originally written for x86 platforms. To evaluate their performance on the SmartNIC, we have manually ported them to Netronome in Micro-C.

Click elements	LoC	Instr.	State	Mem	API	Insights
anonipaddr	93	312	✗	0	7	○●
tcpack	68	142	✗	2	12	○●
udpipencap	87	185	✗	9	15	○●
forcetcp	126	303	✗	3	10	○●
tcpresp	124	452	✗	2	23	○●
tcpgen	108	418	✓	25	22	○●◆◇
aggcounter	95	151	✓	13	3	○●◆◇
timefilter	153	302	✓	22	11	○●◆◇
cmsketch	92	276	✓	5	6	▲△●◆◇
wepdecap	104	332	✓	5	16	▲△●◆◇
iplookup	95	314	✓	23	18	▲△●◆◇
iprewriter	166	331	✓	18	25	▲●◆◇
ipclassifier	372	1860	✓	40	55	▲●◆◇
DNSProxy	974	2438	✓	44	97	▲●◆◇⊕
Mazu-NAT	1266	4127	✓	102	148	○●◆◇⊕
UDPCount	478	992	✓	39	45	○●◆◇⊕
WebGen	469	1221	✓	62	52	○●◆◇⊕

Table 2. The Click programs that we have evaluated, their corresponding LoC, statefulness, the compiled LLVM compute and memory instructions, Click library calls, and the types of offloading insights generated by Clara. Circle: cross-platform prediction; triangle: algorithm identification; solid triangle: reverse porting; solid circle: scale-out factor analysis; diamond: data structure placement; solid diamond: variable reordering and access coalescing; crossed circle: colocation. Solid triangle: Elements dominated by Click data structures, where we apply reverse porting to handle framework differences instead of prediction. The original Click programs are written for the x86 platform, and we have manually ported them to the NIC for evaluation.

Porting. We have used three porting strategies: a) naïve porting, which follows the original logic of the Click NFs faithfully; b) Clara porting, which applies the offloading insights to the porting process; and c) expert emulation, which simulates manual tuning by sweeping relevant parameters for a particular porting decision, and picks the optimal configuration as identified by this exhaustive search. We have also described the classes of offloading insights that apply to each NF in Table 2, as they vary from program to program. The naïve porting strategy further serves as ground truths to evaluate Clara's prediction accuracy. We use the Netronome compiler (NFCC) to generate SmartNIC machine code from the ported NFs.

Methodology. To evaluate the accuracy of Clara's different components, we compare against alternative ML techniques that are commonly used for similar tasks. This includes traditional ML solutions (e.g., kNN, SVM, GBDT, DT), neural networks (e.g., DNN, CNN), and an AutoML solution, TPOT [14, 53]. AutoML solutions are designed to search through different ML pipelines and hyperparameters, with

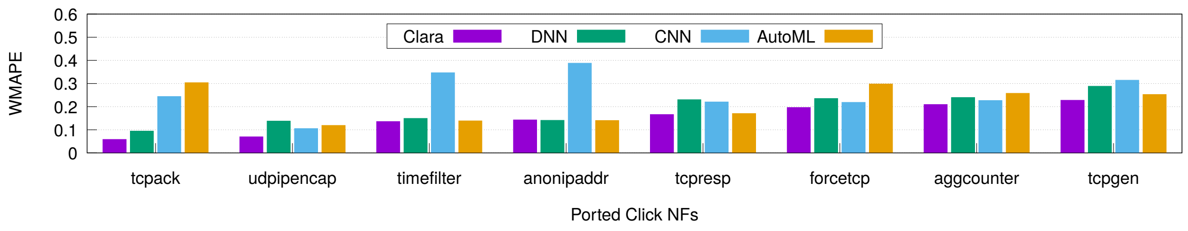


Figure 8. Clara outperforms DNN, CNN and AutoML in instruction prediction.

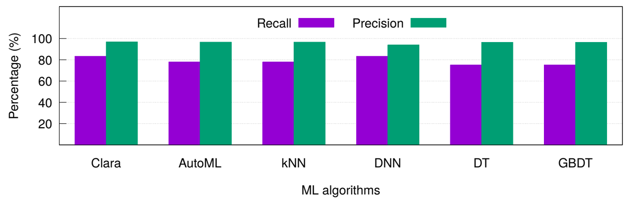


Figure 9. Clara achieves high precision and recall rates for algorithm identification, and is on par with AutoML.

the goal of reducing manual burden in designing and selecting ML models. To evaluate the effectiveness of Clara's offloading insights, we benchmark across different porting strategies. For workload-specific analysis, the training and testing are performed on the same workloads, similar as existing work [49]. A workload specification includes packet sizes, the number of flows, and the IP address distribution.

5.2 Cross-platform prediction

We start by evaluating the accuracy of cross-platform prediction. We primarily evaluate the LSTM+FC model in Clara, which predicts the number of instructions for code that is not reverse ported. As discussed earlier, memory accesses are counted from the number of load/store instructions, and this achieves an accuracy of 96.4%+ for the tested NFs. The LSTM+FC model has converged during training, and achieves similar accuracy on training and testing sets. This indicates that Clara does not overfit narrowly to the seen samples.

Since Clara uses NLP-inspired techniques, we compare its accuracy against other commonly used neural networks (e.g., DNN and CNN) for sentiment analysis and sentence classification tasks [51, 60]. Accuracy is measured by comparing predicted number of compute instructions with the number of instructions in Netronome machine code (by compiling the ported programs using NFCC) on a per-code block basis. The weighted mean-absolute percentage error (WMAPE) of Clara after the training converges is 10.74%. It achieves 6.0%–22.3% across all tested Click NFs. This outperforms all other tested baselines, because Clara specifically takes into account the code sequence information in its prediction, incorporating contextual dependency of the instructions. Figure 8 shows the accuracy of Clara on a set of representative NFs against other baselines.

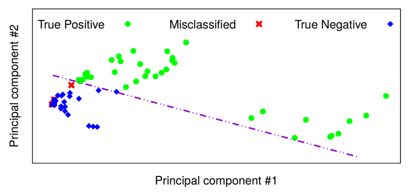
Our comparison against the AutoML solution shows similar results. We have trained TPOT using the same datasets as Clara, and the best ML solution it suggested is an ML pipeline with a random forest regression model. As shown in Figure 8, AutoML does deliver high performance and performs roughly on par with CNN and DNN when aggregated across NFs. After training converges, TPOT achieves an WMAPE of 12.43% on the training set. It achieves WMAPE values ranging from 11.9%-30.3% for real-world NFs. Clara, on the other hand, achieves better performance than TPOT as it is custom designed for the task at hand.

5.3 Algorithm identification

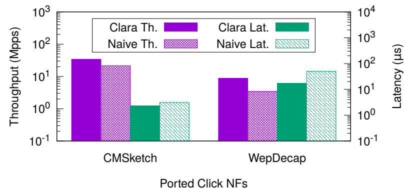
Next, we evaluate the effectiveness of algorithm identification, using precision (TP/TP+FP) and recall (TP/TP+FN) rates as the key metrics. We compare against commonly used ML models for structured data (e.g., kNN, DNN, DT, GDBT) and AutoML. On Netronome, there are acceleration engines for LPM (longest-prefix match), CRC, and other crypto algorithms (e.g., AES, MD5), although typical NFs do not involve cryptographic algorithms. Figure 9 shows that Clara achieves a precision of 96.6% and recall of 83.3% for these accelerators. The AutoML solution has identified a kNN model with different parameters compared to our baseline. We have found that, for this classification task, other models and AutoML have on-par performance, because the accelerator algorithms have very distinct features. Our analysis of the features shows that they intuitively reflect a human understanding of the natures of the algorithms. For instance, a distinctive feature for CRC functions is the high density of bitwise operations, such as `xor`, `and`, and `or`, as well as bitshifts `shl` that iterates over data chunks for checksum computation.

As concrete examples, Clara identifies opportunities to use the LPM accelerator in the 'radixiplookup' element (part of the 'iplookup' NF, which performs IP lookup), and CRC acceleration opportunities in elements like 'rc4' (part of the 'wepdecap' NF, which is a wifi packet decapsulation program). Figure 10(a) further illustrates that Clara effectively separates positive and negative data points, where the X and Y axes represent the two most important feature dimensions extracted via Principal Component Analysis (PCA).

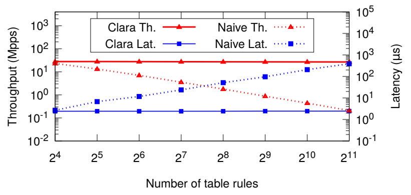
We proceed to evaluate the effectiveness of the Clara algorithm identification insights with sample NFs. Figure 10(b)



(a) PCA analysis

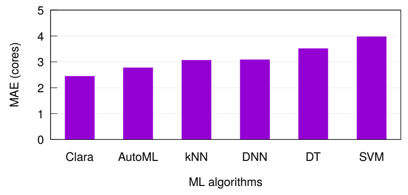


(b) CRC accelerator

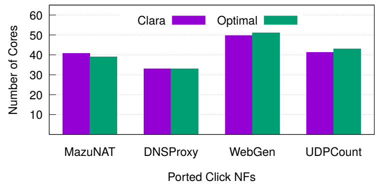


(c) LPM accelerator

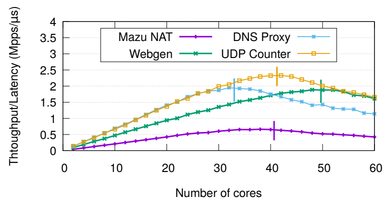
Figure 10. Clara identifies acceleration opportunities accurately, and its suggestions lead to performance improvements.



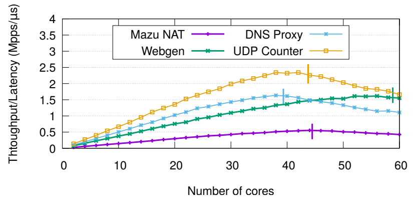
(a) ML alternatives



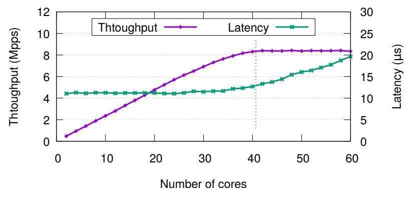
(b) Accuracy on complex NFs



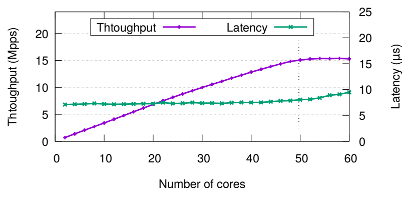
(c) Workload: large flows



(d) Workload: small flows



(e) MazuNAT (Large flows)



(f) Webgen (Large flows)

Figure 11. Clara achieves high accuracy in suggesting the number of cores to use for each NF. The optimal core counts lead to significantly higher performance than naïvely using all cores. Vertical lines highlight Clara's predictions.

shows the benefits of porting the 'count-min sketch' NF and the 'wepdecap' NF to use CRC accelerators, and Figure 10(c) shows the benefits of porting the 'iplookup' NF to use LPM accelerators. As we can see, applying the Clara insights improves the peak throughput of CRC-based NFs by up to 1.6×, and it decreases the latency by up to 25%, compared to the naïve porting that procedurally translate the NF code to the SmartNIC. Similarly, for the 'iplookup' NF, which Clara identified as containing opportunities for LPM acceleration, applying such porting strategies increases throughput and decreases latency by roughly one order of magnitude. These results show that Clara's offloading insights are helpful for improving ported performance.

5.4 Multicore scale-out analysis

Next, we evaluate the effectiveness of Clara in performing multicore scale-out factor analysis in Figure 11. For all NFs, small data structures (e.g., counters) are located in SRAM-based IMEM, and larger ones (e.g., flow-level state) in DRAM-based EMEM with a small SRAM cache. We compare against the ML models that have been used in similar contexts [49] and AutoML, and the results are shown in Figure 11(a). As we can see, the GBDT model in Clara achieves the highest accuracy in this prediction, outperforming other baselines. It also outperforms AutoML, which also identifies GBDT as the most effective model but with different parameters. Figure 11(b) further shows the accuracy of Clara predictions for the most complex Click NFs. On the 60-core Netronomoe NIC, the suggested core counts deviate from the optimal configurations (as obtained by exhaustive benchmarking with all possible configurations) by 1%-6% across NFs. This means that developers can leverage Clara to quickly identify effective multicore configurations for their NFs.

Figures 11(c)-(d) show the throughput/latency ratio curves for different NFs and workloads at different core counts, as well as the suggested configurations by Clara. As we can see, the performance benefits increase at the beginning of the curves, as assigning more cores to these NFs improves throughput with minimum latency increase. However, all curves peak after a certain point, beyond which more cores will simply result in more contention. Moreover, different NFs peak at different core counts, and different workloads also have different optimal configurations. For larger flow sizes, the performance peaks earlier with respect to core counts, as packets mostly produce cache hits. Packet processing therefore utilizes computing engines more effectively. On the other hand, for smaller flows, there are more cache misses, so the cores are not as effectively used. As a result, performance tends to peak later with more cores. Compared to naïvely using all cores, the peak performance as achieved

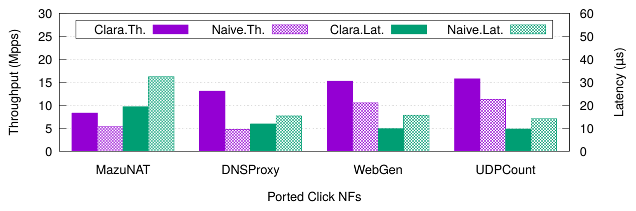


Figure 12. Clara suggests effective NF state placement strategies that lead to higher performance. Workload: Small flows.

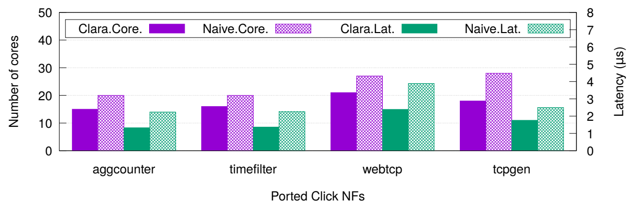


Figure 13. Clara's memory access optimizations work well.

by the optimal core counts is up to 71.1% higher across the tested NFs. Figures 11(e)-(f) further present the latency and throughput figures for two representative NFs, Mazu-NAT and Webgen, in detail. It also highlights Clara's predictions, which are close to the optimal operating points.

5.5 NF state placement

Next, we evaluate the effectiveness of Clara to identify NF state placement strategies to improve performance. As discussed before, Clara analyzes the access frequencies of different data structures on the host platform with a given network trace. It then formulates an ILP constraint solving problem to determine how to place stateful data structures in different memory hierarchies. The baseline solution does not programmatically manipulate state placement, and all data structures are allocated in EMEM.

As shown in Figure 12(a), on average, Clara's placement strategies reduce memory access latency by 33%, and they improve throughput by 89% as compared to the baseline. As a concrete example, in 'UDPCount', small but frequently accessed data structures, such as the ipclassifier and the counter, are allocated in IMEM rather than EMEM. This leads to improved performance over the naïve port. The ILP problem size scales with the number of data structures in a particular NF, which is typically small. ILP solving finishes within a few seconds in all cases.

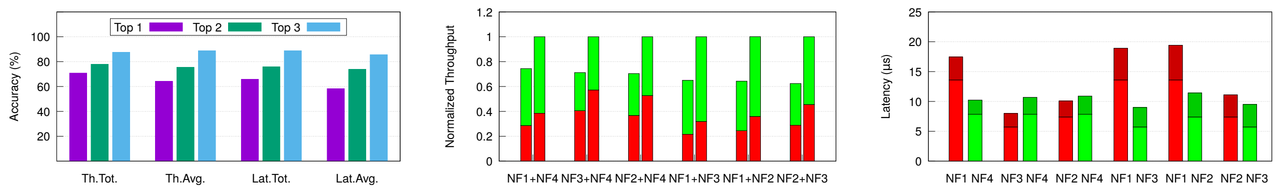
5.6 Memory access coalescing

Next, we evaluate how well Clara can suggest stateful variable packing and access coalescing insights. Figure 13 shows the performance results before and after applying the Clara insights to four Click elements that make extensive use of global variables. We use the number of cores required to saturate the bandwidth as the performance metric. Effective packing leads to fewer memory access stalls, so full bandwidth can be achieved with fewer cores. As concrete examples of Clara output, for the 'tcpgen' NF, one of the clusters suggested by Clara contains source and destination ports; another cluster contains variables for the ACK-processing code paths (e.g., tcp_state, send_next and recv_next); on the other hand, other variables (e.g., 'good_pkt' and 'bad_pkt') are allocated far away from each other as they are never accessed together. Overall, Clara identified five clusters of stateful variables which are not allocated adjacently in the original Click NF. Applying Clara's suggestions to different NFs reduces NF latency by 42%-68%, and this leads to a reduction of core counts by 25%-55%.

5.7 NF colocation

Next, we evaluate the effectiveness of Clara's colocation analysis. Unlike well-studied regression or classification tasks, ranking problems are still under active study, and AutoML solutions currently do not support ranking tasks [1, 15]. Therefore, we focus on analyzing the prediction accuracy of Clara and the NF colocation performance. We have trained four models with different ranking objectives: a) total throughput loss, b) average throughput loss, c) total latency loss, and d) average latency loss. Specifically, a) measures the aggregate colocation throughput and normalizes it by the sum of all individual NF throughputs when they exclusively use the SmartNIC; b) measures the relative throughput loss separately for each NF and averages them. The loss objectives for latency are analogous. After the training converges, we test Clara models on random groups of 1000 synthesized NFs. As shown in Figure 14(a), Clara with the total throughput loss as training objective performs the best, achieving an accuracy of 70+% when suggesting the best (top-1) colocation strategy, and a top-3 ranking accuracy of 85+%.

Further, we show the results on four real-world NFs, for which Clara performs ranking across six pairs of colocation strategies. Clara has correctly ranked all top-3 choices for these NFs. Figure 14(b) shows the throughput degradation due to NF colocation. For each pair, the left bar shows the colocation throughputs (one color for each NF) and the right bar shows the throughputs for exclusive SmartNIC use. As we can see, the top suggestions from Clara have smaller throughput degradation as compared to colocating NFs with lower ranking scores. The throughput degradation varies up to 15% across six colocation strategies. Figure 14(c) further shows the latency results, where the darker regions of each bar denote the latency increase due to colocation for that NF. For this experment, Clara's ranking objective is based on throughput, but as NF contention also results in latency penalty, we see that Clara's suggested strategies are also better for latency metrics. Overall, applying Clara's insights helps developers to identify friendly NF programs for colocated deployments.



(a) Colocation ranking accuracy (b) Throughput loss caused by colocation (c) Latency increase caused by colocation

Figure 14. Clara achieves 70+% accuracy in suggesting the best colocation strategy, and 85+% for suggesting the top three. NF1: Mazu-NAT, NF2: DNSProxy, NF3: UDPCount, NF4: Webgen.

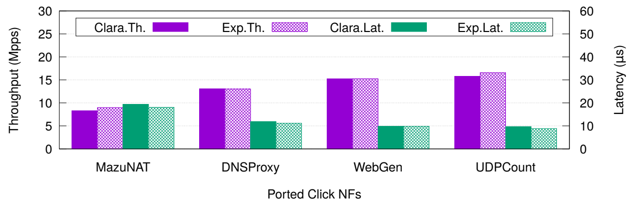


Figure 15. Clara is competitive with 'expert' implementations for NF state placement.

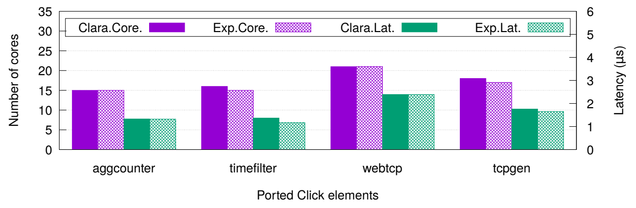


Figure 16. Clara is competitive with 'expert' implementations for memory access coalescing.

5.8 Expert emulation

Next, we explore how well Clara performs against 'expert' implementations. Instead of following the original NF program logic faithfully, an expert programmer that is familiar with SmartNIC details may employ manual tuning to identify the best strategy. Ideally, we could benchmark against preexisting expert ports for comparison, but as SmartNICs are an emerging platform, high-quality SmartNIC ports of Click NFs are scarce. As such, we emulate expert tuning by exhaustively sweeping relevant parameters for a particular porting decision to identify the optimal configuration. As the scale-out analysis already tests against all core configurations, and the NF colocation also measures all possible colocation strategies, these evaluations are already representative of what an expert might achieve. Thus, we focus on NF state placement and memory coalescing use cases.

For NF state placement, we perform an exhaustive search over all possible placement strategies on a per data structure granularity—for each data structure, we try placing it in any memory location that can accommodate its size. For the tested NFs, we find that Clara performs slightly worse but is still on-par with this exhaustive search. As shown in Figure 15, across all cases, Clara's latency is up to 9.7% higher and its throughput is up to 7.6% lower than what is achievable with an exhaustive search. As a concrete example, in 'UDPCount', the optimal placement as identified by an exhaustive search is to place certain state in EMEM rather than IMEM, whereas Clara suggests that all state be placed in IMEM as its size is small. Interestingly, this is because spreading state across two memory regions in this case increases the aggregate bandwidth. Moreover, even if state is placed in DRAM-based EMEM, it is small enough to always fit into its SRAM-based cache. Such information is not captured by Clara's ILP formulation.

For memory coalescing, we emulate expert tuning by an exhaustive search over the relative position of the most frequently visited variables—the total number of variables is too large for an exhaustive analysis. Specifically, we identify variables that are used in the top-3 most frequently triggered code blocks, pack such variables together, and try all possible positions amongst them. As shown in Figure 16, this strategy delivers a small advantage over Clara, although Clara remains competitive on the tested NFs. We found that this is because Clara's optimization focuses on identifying clusters and packing variables within clusters, and this has to use some cutoff threshold to determine some suitable inter-cluster distance. Once cluster boundaries have been determined, clusters of variables are placed independently of each other. On the other hand, exhaustive tuning further takes into account the performance effect due to the relative position of clusters themselves. Some variables, even if they are in different clusters, may still be close to each other if their clusters are packed together.

Overall, this set of experiments demonstrates that the Clara insights are competitive with what an expert might achieve in manual tuning and optimization.

6 Discussion

Other SmartNICs. SmartNIC platforms come in great diversity, but broadly they fall into two categories: a) SoC-based platforms enclose embedded cores for packet processing, and b) FPGA-based platforms are reconfigurable at the gate level for finer-grained programmability. The techniques in Clara target SoC-based platforms with an explicit ISA design. This is because it relies on the LLVM IR of NF programs and the corresponding ISA instructions on the SmartNICs for training and prediction. The current Clara prototype is based on

Netronome Agilio, and other SoC-based SmartNICs include Nvidia BlueField, Marvell LiquidIO, and Broadcom Stingray. In addition, emerging platforms like Pensando DSCs [13] and Fungible DPUs [12] also have ISA-based cores. An interesting exercise would be to evaluate Clara on a wider range of SoC-based platforms. On the other hand, Xilinx Alveo and other FPGA-based platforms would require a different set of techniques for performance analysis as they do not expose an ISA for their processing engines.

NF frameworks. Click is a popular NF framework, but alternatives also exist [54, 55, 69]. For instance, NetBricks [55] is a type-safe framework for Rust-based network functions. eBPF [5] and P4 [10] are also becoming popular for NF prototyping. To handle these frameworks, Clara needs to be augmented to analyze their framework-specific API. It also needs to perform ML training on NF programs written against these frameworks and their corresponding SmartNIC ports. SmartNIC offloading is also gaining popularity beyond network functions [24, 47]. For these programs, Clara training also needs to be enhanced with representative programs in these contexts.

Experience with ML models. Machine learning has seen many applications in computer systems, but often, ML techniques require domain-specific customization to perform well [31]. In Clara, the performance prediction requires developing a customized data synthesis engine and techniques for compacting the vocabulary. Our prior experience of applying LSTM without vocabulary compaction shows much lower performance. Moreover, although deep learning algorithms often show superior performance, more interpretable models [27] may enable new NF tuning and optimization opportunities, as the developers can easily digest the prediction results. For NF developers, SmartNIC platforms contain opaque details that analytical methods cannot easily capture. For SmartNIC vendors, platform specifications are available, but analyzing unported NFs still presents challenges (e.g., performance analysis and tuning) that are best solved with learning techniques.

Partial offloading. Clara analyzes NF offloading scenarios where the entire programs runs on the SmartNIC. Alternatively, a partial offloading scenario might split the NF program between host CPUs and SmartNICs [52, 58]. In order to handle such scenarios, Clara would also need to reason about the communication between SmartNICs and the host, and borrow from work in host performance analysis [41, 49].

7 Related Work

Performance profiling and prediction. Existing tools for performance profiling or prediction analyze programs for GPUs [39, 40, 64], FPGAs [67], mobile SoCs [37], and specialized hardware accelerators [18, 65]. Clara is particularly related to tools that predict cross-platform performance [19, 21, 50, 68, 70], which primarily focus on GPU programs. Clara considers an emerging class of hardware, SmartNICs, and it specifically focuses on NF performance. Our preliminary workshop paper calls out the need for SmartNIC performance prediction [61], but this current work goes beyond performance prediction. It additionally generates *offloading insights* that lead to significant performance improvements.

NF performance. NF performance analysis is an important topic. BOLT [41] analyzes the compute and memory profiles of x86 network functions and predicts their latency characteristics. CASTAN [57] performs worst-case execution time analysis for NFs. SLOMO [49] predicts NF interference using ML on x86 hardware performance counters. In comparison, Clara focuses on *ported* performance on the SmartNIC platform and provides program tuning suggestions.

Automated performance tuning. Clara is inspired by recent work that performs automated performance tuning by exploring the optimization space in a program- and workload-specific manner. TVM [23] is a tensor program compiler that can generate efficient code for ML accelerators, such as GPUs and TPUs. It handles the architectural and runtime opacity of such accelerators by using machine learning to construct device-specific cost models. TVM inherits the concept of "separating the algorithm from the schedule", which was first pioneered in Halide [62], a language and compiler for stencil computations. Clara borrows similar insights. However, TVM and Halide expect the input programs to be working code written against their frameworks. In comparison, Clara focuses on performance tuning suggestions for programs that are yet to be cross-ported.

8 Conclusion

SmartNICs have become a popular offloading platform for network functions (NFs). Existing work has demonstrated the benefits for NF offloading, but the performance characteristics of offloaded programs are opaque prior to porting, and offloading strategies are difficult to reason about. Today, developers need to first cross-port NFs to the SmartNIC, perform workload-specific benchmarks, and then iteratively tune the ported programs to achieve higher performance. Clara enables automated offloading insights by analyzing an unported legacy NF using a combination of program analysis and machine learning techniques. It generates several types of offloading insights for the developer. Our evaluation shows that Clara achieves high prediction accuracy in its analyses, outperforming alternative baseline techniques, and that its offloading insights lead to higher ported performance.

9 Acknowledgments

We thank our shepherd KyoungSoo Park and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by CNS-1801884, CNS-1942219, CNS-2016727, CNS-2106388, CNS-2106751, and CNS-2106199.

References

[1] Autokeras supported tasks. https://autokeras.com/tutorial/overview.

[2] The babel transpiler. https://github.com/babel.

[3] BlueField SmartNIC for Ethernet High Performance Ethernet Network Adapter Cards. https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet.

[4] Clara code repository. https://github.com/824728350/Clara.

[5] eBPF Hardware Offload to SmartNICs. https://www.netronome.com/technology/ebpf.

[6] f2c: From Fortran to C. http://www.netlib.org/f2c/.

[7] A few reasons for using transpilers. https://devopedia.org/transpiler.

[8] LiquidIOII Smart NICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics.html.

[9] Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/products/smartnics/innova-2-flex/.

[10] The P4 language repositories. https://github.com/p4lang.

[11] SmartNIC Overview - Netronome. https://www.netronome.com/products/smartnic/overview/.

[12] The Fungible Data Processing Unit (DPU). https://www.fungible.com/product/dpu-platform/.

[13] The Pensando Distributed Service Platform. https://pensando.io/our-platform/.

[14] The TOPT data science assistant. http://epistasislab.github.io/tpot/.

[15] TPOT supported tasks. http://epistasislab.github.io/tpot/api.

[16] trafgen-A fast, multithreaded network packet generator. http://manpages.ubuntu.com/manpages/xenial/man8/trafgen.8.html.

[17] IEEE P802.3bs 400 GbE Task Force. Adopted Timeline. http://www.ieee802.org/3/bs/, 2018.

[18] M. S. B. Altaf and D. A. Wood. Logca: A performance model for hardware accelerators. *IEEE Computer Architecture Letters*, 14(2):132–135, 2014.

[19] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[20] M. Bysiek, A. Drozd, and S. Matsuoka. Migrating legacy Fortran to Python while retaining Fortran-level performance through transpilation and type hints. In *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, 2016.

[21] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby. CERE: LLVM-based codelet extractor and replayer for piecewise benchmarking and optimization. *ACM Transactions on Architecture and Code Optimization*, 12(1), 2015.

[22] T. Chen and C. Guestrin. XGboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[23] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[24] S. Choi, M. Shahbaz, B. Prabhakar, and M. Rosenblum. λ-NIC: Interactive serverless compute on smartnics. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, 2020.

[25] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predictive modeling. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2017.

[26] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the 2009 ACM Symposium on Operating System Principles (SOSP)*, 2009.

[27] M. Du, N. Liu, and X. Hu. Techniques for interpretable machine learning. *Communications of the ACM*, 63(1):68–77, 2020.

[28] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[29] Y. Fan, Y. Ye, and L. Chen. Malicious sequential pattern mining for automatic malware detection. *Expert Systems with Applications*, 52, 2016.

[30] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[31] S. Fu, S. Gupta, R. Mittal, and S. Ratnasamy. On the use of ML for blackbox system performance prediction. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[32] M. Gallo and R. Laufer. ClickNF: a modular stack for custom network functions. In *2018 USENIX Annual Technical Conference (ATC)*, 2018.

[33] S. Grant, A. Yelam, M. Bland, and A. Snoeren. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM)*, 2020.

[34] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer. Towards understanding the performance of P4 programmable hardware. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[35] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, 1998.

[36] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[37] M. Hill and V. J. Reddi. Gables: A roofline model for mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019.

[38] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[39] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[40] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[41] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea. Performance contracts for software network functions. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr. Metron:NFV service chains at the true speed of the underlying hardware. In *Proceedings of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[43] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[44] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[45] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on*

Operating Systems Principles (SOSP), 2017.

[46] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using iPipe. In *Proceedings of the 2019 ACM SIGCOMM Conference (SIGCOMM)*. 2019.

[47] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[48] V. Livinskii, D. Babokin, and J. Regehr. Random testing for C and C++ compilers with YARPGen. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2020.

[49] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry. Contention-aware performance prediction for virtualized network functions. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM)*, 2020.

[50] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. Grophecy: GPU performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2011.

[51] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao. Deep learning–based text classification: A comprehensive review. *ACM Computing Surveys*, 54(3), 2021.

[52] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[53] R. S. Olson and J. H. Moore. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Proceedings of the Workshop on Automated Machine Learning (AutoML)*, 2016.

[54] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[55] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[56] N. M. Patel. Half-latency rule for finding the knee of the latency curve. *ACM Performance Evaluation Review*, 43:28–29, 2014.

[57] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM)*, 2018.

[58] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A programming system for NIC-accelerated network applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[59] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, et al. Flowblaze: Stateful packet processing in hardware. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[60] S. Poria, E. Cambria, and A. Gelbukh. Deep convolutional neural network textual features and multiple kernel learning for utterance-level multimodal sentiment analysis. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[61] Y. Qiu, Q. Kang, M. Liu, and A. Chen. Clara: Performance clarity for SmartNIC offloading. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[62] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[63] S. Ruder and B. Plank. Learning to select data for transfer learning with bayesian optimization. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[64] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPU applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[65] A. Sriraman and A. Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[66] B. Stroustrup. Cfront: From C++ to C. http://www.softwarepreservation.org/projects/c$_p$lus$_p$lus/index.html#cfront.

[67] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016.

[68] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005.

[69] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP)*, 2019.

[70] J. Zhai, W. Chen, and W. Zheng. Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node. *ACM Sigplan Notices*, 45(5):305–314, 2010.

[71] K. Zhang, D. Zhuo, and A. Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM)*, 2020.