The Case for Performance Interfaces for Hardware Accelerators

Rishabh Iyer[1], Jiacheng Ma[1], Katerina Argyraki[1], George Candea[1], Sylvia Ratnasamy[2]
[1] EPFL, Switzerland [2] UC Berkeley

Abstract

While systems designers are increasingly turning to hardware accelerators for performance gains, realizing these gains is painstaking and error-prone. It can take several person-months to determine if a given accelerator is a good fit for a given piece of code, and accelerators that cost millions of dollars to build can slow down the very systems they were designed to accelerate.

We argue that hardware accelerators *must* come with *performance interfaces*—interfaces that provide usable information about the accelerator's performance behavior just like semantic interfaces do for functionality—to facilitate their correct use. Since accelerators do not provide new functionality and are only useful if they improve system performance, performance interfaces are as integral to their correct use as semantic interfaces.

ACM Reference Format:
Rishabh Iyer, Jiacheng Ma, Katerina Argyraki, George Candea, Sylvia Ratnasamy. 2023. The Case for Performance Interfaces for Hardware Accelerators. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595904

1 Introduction

With the decline of Moore's law, system designers are increasingly reliant on hardware accelerators for performance improvements. From datacenters to hand-held devices, hardware accelerators are used to speed up a wide variety of applications such as machine learning [4, 33, 34, 45], video processing [21, 54], compression, encryption [14, 29] and system infrastructure tasks [5, 22, 26, 36].

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595904

However, designing and building systems that use accelerators *correctly* (i.e., fully extract their performance benefits) is a challenging task, because developers have little to no visibility into an accelerator's expected performance behavior. Every accelerator bakes certain design choices into silicon, such as optimizing throughput over latency [49] or making assumptions about the typical workload [36], and if the software is a poor fit for these choices, running it on the accelerator may result in worse performance [37, 40, 41]. Since building an accelerator requires large teams and takes years, such efforts that miss their mark cost several million dollars [2].

The goal of this work is to facilitate the correct use of accelerators by enabling developers to answer three frequently asked questions about accelerator performance. During the system design phase, when no code has been written, developers should be able to answer: "*What throughput and latency can I expect from this accelerator for my expected workload?*" and "*Which of these accelerators is the best fit for my expected workload?*". Finally, when trying to optimize the performance of an existing implementation, developers should be able to answer: "*What performance can I expect from my code if I offload (a part of it) to this accelerator?*"

The only way developers can answer these questions today is by porting the code to the accelerator, writing representative tests, and benchmarking it—a process that ranges from tedious to impossible. In the best case, this takes several person-months [48, 51, 52] since each accelerator exposes a different Application Programming Interface (API) and Software Development Kit (SDK). In other cases, such as during the system design stage when no code has yet been written, or when the accelerator is built by a manufacturer without access to the code that would be offloaded, it is downright impossible [23].

We argue that accelerators *must* come with *performance interfaces* [30]—constructs that summarize performance behavior just like semantic interfaces summarize functionality—to ensure their correct use. Semantic interfaces (e.g., code documentation, header files) enable developers to quickly and correctly answer the above questions for functionality. Developers routinely use such interfaces to determine the functionality of third-party libraries, which library is best suited to their requirements, or how incorporating a library will affect their system's functionality as a whole. Since an

accelerator's currency is performance—all functionality can be implemented on general-purpose hardware—we argue that performance interfaces are as integral to their correct use as semantic interfaces.

What should performance interfaces for accelerators look like? Who provides them? We propose three candidate representations: (1) natural language text akin to code documentation, (2) simple executable programs akin to specifications, and (3) a Petri-net [47] based model akin to a precise intermediate representation (IR). Each representation provides a different balance between human readability and precision, allowing users to pick the one best suited to their use case. We envision these interfaces being provided by accelerator vendors and shipping with the accelerator, just like semantic interfaces are provided today.

Are such performance interfaces for accelerators feasible? Our initial experiences and discussions with accelerator builders give us three reasons to be optimistic. First, since accelerators are much simpler than today's general-purpose hardware, we can rely on high-fidelity models from the 80s and 90s (summarized in [19]). Second, since a performance interface only describes performance and not functionality [30], it can abstract away all implementation details that are only relevant to functionality, and it could be much simpler than the underlying implementation. Finally, discussions with accelerator builders indicate that they have an intuitive understanding of the factors that impact performance. So, we believe that, for them, writing a performance interface for their hardware is on par (in terms of difficulty) with a software developer writing a semantic interface for their software.

In summary, we argue that performance interfaces for accelerators are both necessary and feasible. Hardware today provides us with *semantic* modularity and interfaces that provide firm foundations for the design of complex systems. In contrast, we do not have equivalent *performance* modularity or interfaces, and that ship has arguably sailed for general-purpose hardware. Accelerators provide a golden opportunity to correct this and ensure that hardware provides reliable performance interfaces that systems software can build upon. We believe that the widespread adoption of such interfaces could be the first step towards a future where we can build systems with well-understood performance, just like types and object-oriented programming enabled us to build programs that were orders of magnitude bigger, better, yet safer than any that came before.

2 System developers are flying blind!

We now describe three scenarios that depict how lack of visibility into accelerators' expected performance, makes designing and building systems that use them a challenging task. We use the term "accelerators" to refer to fixed-function ASICs whose functionality is baked into silicon, such as the TPU [34], the accelerators available on SoC-based SmartNICs [7], Protoacc [36], and Inferentia [4]. We do not consider reprogrammable hardware such as FPGAs and GPUs.

Example #1: System on Chip (SoC) designer

Imagine you are leading the design of the SoC for a SmartNIC. A SmartNIC usually comprises a few general-purpose cores along with hardware accelerators for compression, encryption, RDMA, NVMe virtualization, NVMe-over-fabrics, hardware clock synchronization, etc. These accelerators are usually designed by third-party entities and sold as IP blocks which are put together as an SoC [28].

The early stages of any SoC design requires answering the question: "*Which accelerator implementations (IP blocks) should my SoC include and how big (area) must each be?*"

There is no good way to answer this question today [27]. Simulating each configuration by porting example workloads is typically infeasible or outright impossible. In the best-case scenario, the search space of all possible IP block combinations is too large. In the worst case, you might not even have access to the code you are building the accelerator for. For instance, cloud providers (e.g., AWS, Azure) who have custom accelerators manufactured for them typically do not share their proprietary code with the manufacturer (e.g., Intel, NVIDIA).

As a result, SoC designers typically rely on heuristics and accumulated wisdom to decide SoC configurations [27]. This can lead to expensive mistakes because designing an SoC typically takes 2-3 years and costs several million dollars [2].

Example #2: Infrastructure stack developer

Imagine you are the engineer responsible for managing the RPC stack at an enterprise datacenter. Currently, your stack runs on commodity servers, but you are considering offloading it to an accelerator. Your candidate hardware platforms include new servers with accelerators for RPC serialization/deserialization, such as Protoacc [36] or Optimus Prime [49] or one of several SmartNICs.

Ideally, you want to answer questions such as: "*Which available accelerator/SmartNIC offers me the best performance per dollar?*", "*How many CPU cores can I save with an offloaded stack?*", and "*What is the performance impact of offloading different portions of my stack?*"

The only way to gain any intuition with respect to the above questions is to purchase every candidate accelerator, port the code, and benchmark it—a process that can take several person-months per accelerator [48, 51, 52]. Most commercial accelerators only describe the *functionality* they accelerate [7] and the few that describe expected performance

only provide upper/lower bounds (e.g., maximum sustainable throughput) or performance for standard benchmarks (e.g., OVS for SmartNICs) [43]. Even once you purchase the accelerator, you do not gain access to any additional performance details; you must port your code and benchmark it. This is typically a painstaking process given that each accelerator exposes a vendor-specific API and SDK [52].

We want to emphasize that acquiring *any* accelerator and blindly offloading the code is not only suboptimal but can also degrade system performance. Every accelerator trades off generality for efficiency and bakes certain assumptions about the workload into silicon. Among RPC data (de)serialization accelerators for example, Optimus Prime is best suited for small data objects (<= 300B), while Protoacc is best suited for larger data objects (>= 4KB) [36]. For workloads comprising small data objects (e.g., short strings), Protoacc can perform worse than a regular Xeon due to the cost of transferring the data to and from the accelerator [36]. Thus, system designers cannot blindly deploy any particular accelerator and hope to reap performance benefits.

Example #3: Compilers for accelerators

Just like the developers above, toolchains that use accelerators also suffer from the lack of performance visibility. We demonstrate this using the example of TVM [12], a widely used state-of-the-art compiler for machine learning.
To optimize the latency of deep-learning inference, TVM "auto-tunes" [13] code to available accelerators using a two-step process. In the first step, the compiler extensively profiles the accelerator using multiple candidate instruction sequences and extracts a program-specific cost model. In the second, the compiler leverages a learning-based search algorithm that generates optimized code based on the extracted model.

However, autotuning code is time-consuming and is bottlenecked by the first (profiling) step. This is because the compiler has no way to quickly and accurately answer the question: "*What latency can I expect when running this sequence of instructions on the accelerator?*". The compiler is forced to treat the accelerator as a black box and profile it either by running cycle-accurate simulations or by synthesizing and running the code directly on the accelerator. Since both the above options are known to be slow [6] autotuning code can take several minutes to a few hours.

In summary, the design (examples #1 and #2) and implementation (example #3) of systems that use accelerators is impaired by the lack of usable information about their performance behavior. We believe that the status quo is equivalent to writing software without semantic interfaces. With accelerators expected to become ubiquitous in the near future [44, 61], this status quo must change.

3 Performance Interfaces for Accelerators

We now describe our proposal for what performance interfaces for accelerators should look like. Throughout this section, we use 4 open-source accelerators as running examples: a Bitcoin miner [8], a JPEG decoder [35], an RPC message (de)serializer from Google named Protoacc [36] and the VTA deep-learning accelerator [42].

We propose three candidate representations for performance interfaces: (1) natural language (English), (2) simple, executable programs (Python), and (3) a Petri net [47]. We picked these three representations as counterparts to widely used representations for semantic interfaces: we expect the English text to resemble semantic interfaces extracted from code documentation by frameworks such as Doxygen [18] and Javadoc [32], the Python programs to resemble functional specifications, and the Petri nets to resemble an intermediate representation like LLVM.

Each representation provides a different balance between human readability (being smaller and simpler than the implementation) and precision (being accurate enough to reason about the several factors that impact performance). Natural language interfaces are the easiest to read but can only describe a few key performance properties. Programs are harder for developers to read but can provide more details such as the specific request types the accelerator is best suited for. Finally, Petri nets are not human-readable but can accurately (< 2% average error in our experiments) predict the accelerator's throughput and latency for arbitrary workloads.

We expect the first two representations to be most useful during the system design stage, when either no code has been written (example #1 in §2) or when developers and tools do not have access to the accelerator (example #2). We expect the Petri-net IR to be most useful during the system implementation and optimization stages (example #3).

Natural language interfaces

Fig. 1 illustrates three interfaces as natural language text for the JPEG decoder, Bitcoin miner, and Protoacc. All three interfaces describe only the key performance aspects at a high level. For the JPEG decoder, the latency is inversely proportional to the compression rate ($\frac{output\_image\_size}{input\_image\_size}$) since the accelerator must perform a similar computation on every input byte. For the Bitcoin miner, a user can simultaneously configure both the latency and size of the accelerator using the parameter `Loop`. This is because the accelerator implements a SHA-256 hash calculation with `Loop` controlling the extent to which the hash function's loop is unrolled in hardware. Finally, Protoacc's throughput decreases as the degree of nesting within a message increases, since each nesting involves a pointer-chasing operation.

Latency is inversely proportional to the input image's compression rate

---

Latency (cycles) is equal to the configuration parameter `Loop`. However, the area occupied by accelerator grows inversely with `Loop`.

Throughput decreases as the degree of nesting in a messages increases

Figure 1: **Example interfaces as English text for the JPEG decoder, Bitcoin miner, and Protoacc (top to bottom).**

This simple representation of performance has two benefits: (1) It helps developers understand how performance varies across inputs for what are otherwise black boxes. For instance, the SoC designer in example #1 could use the interface of the Bitcoin miner to decide how much area on the chip she should allocate to it while knowing how that impacts overall latency. (2) It offers the lowest barrier to adoption. In all our discussions with accelerator builders, they were able to produce such interfaces within seconds. We envision such interfaces being similar to the Big-*O* notation, which, despite providing only asymptotic bounds, continues to be used even in today's microsecond-scale systems (e.g., Redis [55]).

Interfaces as executable programs

Figures 2,3 provides examples of latency interfaces as Python programs for two accelerators: the JPEG decoder and Protoacc. Just like a specification, the programs take the same inputs as the accelerator—an image and the message to be serialized, respectively—but instead of describing how the accelerator computes the correct output, they only describe the accelerator's latency and throughput as a function of the input.

Such interfaces provide a more precise characterization of accelerator performance than the natural language interfaces above. For example, the JPEG decoder's latency interface not only provides the constant factors in the latency expression but also indicates that latency has an upper bound ($size * 136.5$). Similarly, Protoacc's throughput interface tells us the achievable throughput for each of its read and write stages, allowing developers to identify which messages are bottlenecked by each stage. Providing a closed-form formula for Protoacc's latency is hard. This is because the read and write latencies for a single message overlap in complex, message-dependent ways. Hence the latency interfaces in Fig. 3 only provide an upper and lower bound, which is still much better than no information at all.

Interfaces as Python programs can provide reasonably accurate performance predictions. We evaluated JPEG's latency and throughput interfaces using 1500 random images and observed an average (maximum) prediction error of 2.1% (10.3%) and 2.2% (11.2%) respectively. Similarly, when evaluating Protoacc's throughput and latency interfaces using 32 message formats from its test suite, we observed an average

```python
def latency_jpeg_decode(img):
    size = img.orig_size/64
    return max(size*136.5,
        size/64*((5/img.compress_rate)*3+6)*1.5)

def tput_jpeg_decode(img):
    # Images are processed one-by-one
    return 1/latency_jpeg_decode(img)
```

Figure 2: **Interfaces as Python programs for the JPEG decoder.**

```python
def read_cost(msg):
  cost=0
  for sub_msg in msg:
      cost+= read_cost(sub_msg)
  return cost+6 + avg_mem_latency*2 + (4+
    avg_mem_latency) * ceil(msg.num_fields/32)

def tput_protoacc_ser(msg):
  sub_msg_cost = 0
  for sub_msg in msg:
      sub_msg_cost+= read_cost(sub_msg)
  read_tput = 1/((4+avg_mem_latency)*ceil(msg.
    num_fields/32) + sub_msg_cost)
  write_tput = 1/(5+msg.num_writes)
  return min(read_tput,write_tput)

def min_latency_protoacc_ser(msg):
  return (5+msg.num_writes)*avg_mem_latency

def max_latency_protacc_ser(msg):
  sub_msg_cost = 0
  for sub_msg in msg:
      sub_msg_cost += read_cost(sub_msg)
  return min_latency_protoacc_ser(msg) + (4+
    avg_mem_latency)*ceil(msg.num_fields/32) +
    sub_msg_cost
```

Figure 3: **Interfaces as Python programs for Protoacc.**

(maximum) error of 5.9% (13.3%) for throughput, while the latency was always within the predicted bounds.

We envision such interfaces being used during the system design stage to (1) quickly compute the speedup for the computation offloaded to the accelerator and (2) compare multiple accelerator implementations that provide identical functionality. For instance, the infrastructure stack designer in example #2 could use such interfaces to compute which RPC data transformation accelerator or SmartNIC gives her the maximum speedup per dollar for her expected workload, by running the corresponding Python programs on representative RPC objects.

Formal Petri net interfaces

With this representation, we seek to create a "performance IR" for accelerators, i.e., an abstraction that enables tools (as opposed to humans) to precisely reason about accelerator performance, just like how LLVM IR [39] enables compilers to reason about code semantics. We cannot reuse existing IRs because they are designed for software's sequential execution model. In contrast, accelerators typically comprise multiple pipeline stages operating in parallel, leading to complexities such as backpressure, internal queueing, and asynchronous processing.

Accelerator	Avg (max) prediction error		Complexity w.r.t implem.
	Latency	Throughput	
JPEG	0.09% (0.50%)	0.09% (0.51%)	2.5%
VTA	1.49% (9.3%)	1.44% (8.55%)	2.6%

Table 1: **Prediction accuracy and complexity of interfaces as Petri nets. Complexity is measured as the ratio of LOC in the Petri net as compared to the LOC in the implementation.**

We therefore choose to represent the performance IR as Petri nets [47], a class of graphs designed for modeling parallel asynchronous systems. Petri nets consist of four elements: places, tokens, transitions, and edges. Places represent data queues (e.g., buffers, registers), tokens represent individual data units flowing through the system, transitions represent data transformations (e.g., an ALU computation) and edges connect places and transitions. The flow of data through the net is determined by the firing of its transitions. Transitions fire whenever all their input places have sufficient tokens; when they do, they consume tokens from their input places and produce tokens in their output places.

We model accelerators using Petri nets as follows: For each processing element in the accelerator, we write a transition function that a) captures its delay, i.e., how long it takes to process input data (tokens) and b) transforms the tokens to ensure that downstream transition functions can accurately compute their delays. Since multiple transitions can fire simultaneously, and edges capture inter-element dependencies, our Petri net-based performance IR accurately captures both the accelerator's parallel execution model and factors such as internal queuing and backpressure. Thus, the Petri net represents a circuit that is "performance equivalent" to the accelerator's circuit.

We manually derived Petri net interfaces for two accelerators: the JPEG decoder and VTA. We evaluated their prediction accuracy using 50 random images and 1500 random code sequences respectively (Table 1). We observed that the JPEG decoder's Petri net predicted both latency and throughput with an average and maximum error of 0.09% and 0.5% respectively, both of which are about 20× lower than the corresponding errors of the Python programs. For the more complex VTA, the average and maximum prediction error is higher, about $\approx 1.5\%$ and $\approx 9\%$ respectively. These errors arise due to us deliberately cutting corners to make the manual derivation of the Petri nets easier. We are confident that with extra effort, the Petri nets can be fully precise.

To show how the Petri net-based IR can be immediately useful, we added support for it in TVM's auto-tuning engine and used it to profile VTA for the 1500 code sequences. We observed that the Petri-net interfaces lead to a maximum (minimum) speedup of $1,312\times$ ($2.1\times$) over state-of-the-art cycle-accurate simulation [63]. This speedup stems from the fact that the Petri net does not concern itself with the semantics of the accelerator, but rather it only aims to have the same performance properties. This makes it much simpler than the accelerator's internal circuit (Table 1) and enables it to run much faster.

A natural question at this point is: "Why propose the previous representations when we can have a precise IR?" The answer is that Petri nets are hard to understand for a developer unfamiliar with the accelerator's internal circuitry. Such developers (which we expect to be the majority) can only use tools to analyze or run the Petri net. In contrast, developers can quickly eyeball the human-readable interfaces in English or Python and tell how performance will vary across inputs.

In summary, we propose three representations for accelerator performance interfaces that enable developers and tools to reason about latency and throughput at different levels of precision. Our initial experiences make us optimistic that such interfaces are both useful and feasible. However, there remain several challenges that must be overcome for this vision to become a reality; we discuss these further in §5.

4 Related Work

Performance modeling: While traditional approaches to summarizing performance (such as upper bounds and benchmark scores) are useful, they are ill-suited to being performance *interfaces* because they cannot tell developers what to expect for *their* code and workload. For instance, an accelerator like Optimus Prime can sustain a maximum throughput of 33 Gbps, but this drops to 14 Gbps for realistic workloads. Similarly, while standardized benchmark suites have been widely used in the computer architecture (e.g., SPEC [59]), databases (e.g., TPC-C [62]) and machine learning (e.g., Dawnbench [17]) communities, developers must extrapolate how similar their code is to each program in the benchmark suite if they are to know what performance to expect [50].

We were heavily inspired by more recent work on performance modeling of both software [15, 16, 24, 25, 30, 31, 38, 52, 56, 64, 65] and hardware [9, 10, 19, 20, 27, 28, 37]. In particular, we borrowed the notion of performance interfaces as programs from PIX [30] and Freud [56], although both use such interfaces for software running on general-purpose hardware and neither can reason about throughput.

Better compilers for accelerators: To eliminate the effort of porting code across accelerators, researchers have proposed several compilers that automatically generate code that runs on different accelerators [12, 46, 53, 57]. We see such compilers as being complementary to interfaces because they are only useful once the developer has both an implementation and access to the accelerator, and cannot

answer questions during the system design stage. Furthermore, compilers themselves can benefit from performance interfaces to generate better code, as shown in example #3.

5 Open Questions

Are performance interfaces for accelerators feasible as accelerator complexity increases? Our initial experience with Petri nets makes us optimistic. After all, VTA is a fairly complex accelerator with internal queuing, parallelism, and deep pipelines, and our Petri net model was able to summarize its throughput and latency for arbitrary instruction sequences with an average prediction error of $< 2\%$.

That said, there are still several open challenges, most of which do not arise from the accelerator's internal circuitry, but rather from how it interacts with other hardware structures, such as the TLB and interconnects. For example, since co-processors like Protoacc access memory via the TLB [36], the Petri net model would need to include the TLB state to be able to reason precisely about memory access latencies. Similarly, a Petri net for a SmartNIC will likely need to include a model of the interconnect, since it can have a significant impact on performance [41]. One possible solution to this challenge could be to develop individual Petri nets for such components once and reuse them across multiple accelerators.

How should accelerator vendors produce performance interfaces for their hardware? As a start, we believe that accelerator designers can manually produce performance interfaces, akin to how software developers are responsible for writing semantic interfaces. We found that accelerator builders usually possess an intuitive understanding of the factors that impact the performance of their hardware and can immediately come up with natural language interfaces that describe whether their accelerators are memory-/compute-bound and what aspects of the input workload are likely to lead to performance variability. Producing the other two representations is harder and typically took us ≈ 2 person-days to extract and test, once we had fully understood the accelerator's implementation. Contrary to our expectations, the Python programs took as long as the Petri nets, since it was often hard to decide what level of abstraction to expose in the program while translating a circuit into a Petri net was a fairly mechanical process.

That said, we believe that building tools that can automatically extract interfaces as Petri nets or Python programs from accelerator implementations is a promising direction for future work. While there has been recent work on tools for automatic (semantic) analysis of Verilog code [1, 3, 11, 58, 60], using these tools to analyze the performance of production ASICs remains an open question.

How can one use performance interfaces to reason about end-to-end performance? Predicting the impact of a *partial* offload on end-to-end application performance is a hard problem. To do so accurately, one must take into account factors such as how the original codebase will need to change, data movement costs, etc. which will vary from application to application. Merely plugging performance interfaces into the original code is not sufficient; while the interface can return performance, it will not return a semantically meaningful response and may cause the calling code to behave arbitrarily.

Nevertheless, we believe that *executable* performance interfaces such as Python programs and Petri nets can enable developers to answer this question with a little extra effort. A strawman solution would work as follows: The application is first run with a software implementation of the accelerator's API and all requests and responses are saved. The application is then re-run with a simple simulator that spins idly for the latency computed by the interface for the input request and then returns the correct, saved response. Since accelerator invocations are typically pure functions, such a strawman should work, at least for deterministic applications.

6 Conclusion

We argue that hardware accelerators should ship with interfaces that provide usable information about their expected performance behavior and that such interfaces are as integral to their correct use as semantic interfaces that describe functionality. We believe that such interfaces can be the first step toward a future where we build systems with well-understood performance, just like types and object-oriented programming enabled us to build programs that were orders of magnitude bigger, yet safer than any that came before.

However, any such interfaces will have to come from hardware vendors, and they will only do so if there is sufficient pressure from their primary customers, i.e., the systems community. Hence, we invite the community to actively pursue research on what performance information these interfaces should expose, how they can be used, and how they can be provided by hardware vendors.

7 Acknowledgements

We thank our shepherd Jialin Li and the HotOS reviewers for their detailed feedback that significantly improved the paper. We are also grateful to the many people whose inputs helped shape our thoughts—Thomas Bourgeat, Mahyar Emami, Narek Galstyan, Siddharth Gupta, Sagar Karandikar, Sahand Kashani, and James Larus.

References

[1] Andraus, Z. S., Liffiton, M. H., and Sakallah, K. A. Reveal: A Formal Verification Tool for Verilog Designs. In *Logic for Programming, Artificial Intelligence, and Reasoning* (2008).

[2] The Economics of ASICs. https://www.electronicdesign.com/technologies/embedded-revolution/article/21808278/ensilica-the-economics-of-asics-at-what-point-does-a-custom-soc-become-viable.

[3] Athalye, A., Kaashoek, M. F., and Zeldovich, N. Verifying Hardware Security Modules with Information-Preserving Refinement. In *Symp. on Operating Sys. Design and Implem.* (2022).

[4] AWS Inferentia Accelerators for Deep Learning Inference. https://aws.amazon.com/machine-learning/inferentia/.

[5] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[6] Beamer, S. A Case for Accelerating Software RTL Simulation. In *IEEE Micro* (2020).

[7] NVIDIA Bluefield-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[8] Open Source Bitcoin Miner. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner.

[9] Boran, N. K., Yadav, D. K., and Iyer, R. Performance Modelling and Dynamic Scheduling on Heterogeneous-ISA Multi-core Architectures. In *Intl. Symp. on VLSI Design and Test* (2019).

[10] Boran, N. K., Yadav, D. K., and Iyer, R. Classification based scheduling in Heterogeneous ISA Architectures. In *Intl. Symp. on VLSI Design and Test* (2020).

[11] Chattopadhyay, S., Lonsing, F., Piccolboni, L., Soni, D., Wei, P., Zhang, X., Zhou, Y., Carloni, L. P., Chen, D., Cong, J., Karri, R., Zhang, Z., Trippel, C., Barrett, C. W., and Mitra, S. Scaling Up Hardware Accelerator Verification using A-QED with Functional Decomposition. In *Formal Methods in Computer Aided Design* (2021).

[12] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Symp. on Operating Sys. Design and Implem.* (2018).

[13] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems* (2018).

[14] Chiosa, M., Maschi, F., Müller, I., Alonso, G., and May, N. Hardware Acceleration of Compression and Encryption in SAP HANA. In *Intl. Conf. on Very Large Databases* (2022).

[15] Coppa, E., Demetrescu, C., and Finocchi, I. Input-sensitive Profiling. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

[16] Coppa, E., Demetrescu, C., Finocchi, I., and Marotta, R. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. In *Intl. Symp. on Code Generation and Optimization* (2014).

[17] DAWNBench: An End-to-End Deep Learning Benchmark and Competition. https://dawn.cs.stanford.edu/benchmark/.

[18] Doxygen. https://www.doxygen.nl.

[19] Eeckhout, L. Computer Architecture Performance Evaluation Methods. In *Synthesis Lectures on Computer Architecture* (2010).

[20] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A Performance Counter Architecture for Computing Accurate CPI Components. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

[21] Facebook: Video transcoding with Mount Shasta. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

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

[23] Formal Methods Only Solve Half My Problems. https://brooker.co.za/blog/2022/06/02/formal.html.

[24] Fu, S., Gupta, S., Mittal, R., and Ratnasamy, S. On the use of ML for blackbox system performance prediction. In *Symp. on Networked Systems Design and Implem.* (2021).

[25] Goldsmith, S., Aiken, A., and Wilkerson, D. S. Measuring empirical computational complexity.

[26] Google-Intel Infrastructure Processing Unit (IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html.

[27] Hill, M., and Janapa Reddi, V. Gables: A Roofline Model for Mobile SoCs. In *Intl. Symp. on High-Performance Computer Architecture* (2019).

[28] Hill, M. D., and Reddi, V. J. Accelerator-level parallelism. In *Communications of the ACM* (2021).

[29] Intel QAT: Accelerating Data Compression and Encryption. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[30] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symp. on Networked Systems Design and Implem.* (2022).

[31] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance Contracts for Software Network Functions. In *Symp. on Networked Systems Design and Implem.* (2019).

[32] Javadoc. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

[33] Jouppi, N. P., Yoon, D. H., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P. C., Ma, X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. A. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *Intl. Symp. on Computer Architecture* (2021).

[34] Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Intl. Symp. on Computer Architecture* (2017).

[35] High throughput, pipelined JPEG decoder. https://github.com/ultraembedded/core_jpeg.

[36] Karandikar, S., Leary, C., Kennelly, C., Zhao, J., Parimi, D., Nikolic, B., Asanovic, K., and Ranganathan, P. A Hardware Accelerator for Protocol Buffers. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2021).

[37] Kim, M. A., and Edwards, S. A. Computation vs. Memory Systems: Pinning down Accelerator Bottlenecks. In *Intl. Symp. on Computer Architecture* (2010).

[38] Krude, J., Rüth, J., Schemmel, D., Rath, F., Folbort, I., and Wehrle,

K. Determination of Throughput Guarantees for Processor-based SmartNICs. In *Intl. Conf. on Emerging Networking Experiments and Technologies* (2021).

[39] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Intl. Symp. on Code Generation and Optimization* (2004).

[40] Liu, J., Maltzahn, C., Ulmer, C. D., and Curry, M. L. Performance Characteristics of the BlueField-2 SmartNIC. *arxiv.org/cs* (2021).

[41] Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., and Gupta, K. Offloading Distributed Applications onto SmartNICs Using IPipe. In *ACM SIGCOMM Conf.* (2019).

[42] Moreau, T., Chen, T., Vega, L., Roesch, J., Yan, E. Q., Zheng, L., Fromm, J., Jiang, Z., Ceze, L., Guestrin, C., and Krishnamurthy, A. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. In *IEEE Micro* (2019).

[43] Netronome Agilio OVS Benchmarking. https://www.netronome.com/media/documents/WP_OVS-TC_40G.pdf.

[44] Nider, J., and Fedorova, A. S. The last CPU. In *Workshop on Hot Topics in Operating Systems* (2021).

[45] Norrie, T., Patil, N., Yoon, D. H., Kurian, G., Li, S., Laudon, J., Young, C., Jouppi, N. P., and Patterson, D. A. Google's Training Chips Revealed: TPUv2 and TPUv3. In *IEEE Hot Chips Symposium* (2020).

[46] Pereira, F., Matos, G., Sadok, H., Kim, D., Martins, R., Sherry, J., Ramos, F. M. V., and Pedrosa, L. Automatic Generation of Network Function Accelerators using Component-based Synthesis.

[47] Peterson, J. L. Petri nets. In *ACM Computing Survey* (1977).

[48] Phothilimthana, P. M., Liu, M., Kaufmann, A., Peter, S., Bodík, R., and Anderson, T. E. Floem: A Programming System for NIC-Accelerated Network Applications. In *Symp. on Operating Sys. Design and Implem.* (2018).

[49] Pourhabibi, A., Gupta, S., Kassir, H., Sutherland, M., Tian, Z., Drumond, M. P., Falsafi, B., and Koch, C. Optimus Prime: Accelerating Data Transformation in Servers. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2020).

[50] The Problem with Benchmarking Hardware. https://semiengineering.com/the-problem-with-benchmarks/.

[51] Qiu, Y., Kang, Q., Liu, M., and Chen, A. Clara: Performance Clarity for SmartNIC Offloading. In *ACM Workshop on Hot Topics in Networks* (2020).

[52] Qiu, Y., Xing, J., Hsu, K.-F., Kang, Q., Liu, M., Narayana, S., and Chen, A. Automated SmartNIC Offloading Insights for Network Functions. In *Symp. on Operating Systems Principles* (2021).

[53] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. P. Halide: a Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Intl. Conf. on Programming Language Design and Implem.* (2013).

[54] Ranganathan, P., Stodolsky, D., Calow, J., Dorfman, J., Hechtman, M. G., Smullen, C., Kuusela, A., Laursen, A. J., Ramirez, A., Wijaya, A. A., Salek, A., Cheung, A., Gelb, B., Fosco, B., Kyaw, C. M., He, D., Munday, D. A., Wickeraad, D., Persaud, D., Stark, D., Walton, D., Indupalli, E., Perkins-Argueta, E., Lou, F., Wu, H. K., Chong, I. S., Jayaram, I., Feng, J., Maaninen, J., Lucke, K. A., Mahony, M., Wachsler, M. S., Tan, M., Penukonda, N., Dasharathi, N., Kongetira, P., Chauhan, P., Balasubramanian, R., Macias, R., Ho, R., Springer, R., Huffman, R. W., Foss, S., Bhatia, S., Gwin, S. J., Sekar, S. K., Sokolov, S. N., Muroor, S., Rautio, V.-M., Ripley, Y., Hase, Y., and Li, Y. Warehouse-Scale Video Acceleration: Co-design and Deployment in the Wild. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2021).

[55] Redis Documentation including Big-O Time Complexity for each command. https://redis.io/commands.

[56] Rogora, D., Carzaniga, A., Diwan, A., Hauswirth, M., and Soulé, R. Analyzing System Performance with Probabilistic Performance Annotations. In *ACM EuroSys European Conf. on Computer Systems* (2020).

[57] Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J. K., Shao, C., Mishra, A., and Esmaeilzadeh, H. From high-level deep neural models to FPGAs. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2016).

[58] Singh, E., Lonsing, F., Chattopadhyay, S., Strange, M., Wei, P., Zhang, X., Zhou, Y., Chen, D., Cong, J., Raina, P., Zhang, Z., Barrett, C., and Mitra, S. A-QED Verification of Hardware Accelerators. In *Design Automation Conference* (2020).

[59] Standard Performance Evaluation Corporation (SPEC). https://spec.org/benchmarks.html.

[60] SymbiYosys. https://github.com/YosysHQ/sby.

[61] Tork, M., Maudlej, L., and Silberstein, M. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2020).

[62] TPC-C: An On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc.

[63] Verilator. https://www.veripool.org/verilator.

[64] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* (2008).

[65] Zaparanuks, D., and Hauswirth, M. Algorithmic Profiling. In *Intl. Conf. on Programming Language Design and Implem.* (2012).