Performance Interfaces for Hardware Accelerators

Jiacheng Ma, Rishabh Iyer, Sahand Kashani, Mahyar Emami, Thomas Bourgeat, George Candea
EPFL, Switzerland

Abstract

Designing and building a system that reaps the performance benefits of hardware accelerators is challenging, because accelerators provide little concrete visibility into their expected performance. Developers must invest many person-months into benchmarking to determine if their system would indeed benefit from using a particular accelerator. This must be done carefully, because accelerators can actually hurt performance for some classes of inputs, even if they help for others [53].

We demonstrate that it is possible for hardware accelerators to ship with *performance interfaces* that provide actionable visibility into their performance, just like semantic interfaces do for functionality. We propose an *intermediate representation* (IR) for accelerator performance that precisely captures all performance-relevant details of the accelerator while abstracting away all other information, including functionality. We develop *a toolchain* (ltc) that, based on the proposed IR, automatically produces human-readable performance interfaces that help developers make informed design decisions. ltc can also automatically produce formal *proofs of performance properties* of the accelerator, and can act as a *fast performance simulator* for concrete workloads.

We evaluate our approach on accelerators used for deep learning, serialization of RPC messages, JPEG image decoding, genome sequence alignment, and on an RMT pipeline used in programmable network switches. We demonstrate that the performance IR provides an accurate and complete representation of performance behavior, and we describe a variety of use cases for ltc and the resulting performance interfaces.

The code for ltc is open-source and freely available at [68].

1 Introduction

From datacenters to hand-held devices, modern systems increasingly rely on hardware accelerators to speed up a variety of tasks, such as machine learning [4, 48, 49, 64], video processing [28, 73], compression, encryption [17, 40], communication [29, 53], and even system infrastructure tasks [5, 32].

However, building a system that uses accelerators correctly—i.e., that fully extracts their performance benefits—remains a challenging task, because software engineers have little to no visibility into an accelerator's expected performance behavior. Every accelerator bakes design choices into silicon, such as specific throughput-vs-latency trade-offs [70] or assumptions about the workload [53], and if the software is a poor fit for these choices, acceleration will offer few benefits or even make performance worse [55, 59, 60].

This lack of visibility into expected performance hampers system developers in all three stages of system development: design, implementation, and deployment.

First, during the design stage, what functionality (if any) to offload, and which accelerators to use, is not obvious. Consider the offloading of (parts of) an RPC stack to an accelerator, where the candidates are RPC serializers/deserializers like ProtoAcc [53] and Optimus Prime [70], or one of several SmartNICs. Software engineers need to know what latency and throughput they can expect from each candidate accelerator, given their code and workload. Then they can decide which one offers the best price–performance ratio, before investing in thousands of new chips and refactoring the RPC stack. To answer these questions today, one needs to purchase every candidate accelerator, port the code, and benchmark them together—performance depends not only on the accelerator but also on the code and workload. For example, Optimus Prime is best suited for small data objects ($\leq$300B), while ProtoAcc is best suited for larger data objects ($\geq$4KB) [53], but this does not transpire at all from vendors' datasheets. Blindly offloading to *any* accelerator is not an option either, because this can end up degrading system performance. For instance, for workloads comprising long strings, ProtoAcc can perform worse than a regular Xeon server, because the accelerator is bottlenecked by memory-intensive operations [53].

Second, in the implementation stage, software engineers want to know how they can best optimize their code for the chosen accelerator. Ideally, tools like compilers should answer such questions quickly and automatically, but compilers too are hampered by the lack of visibility into accelerator performance. For instance, the TVM compiler [15]—a widely used compiler for deep learning models—takes several hours to optimize code for a target accelerator [16, 58]. This is because the compiler cannot figure out quickly and accurately what latency can be expected when running a specific sequence of instructions on the accelerator. So it generates multiple variants of the code and profiles them on the accelerator itself (or on slower cycle-accurate simulators [7] when the accelerator is not available) to pick the optimal one. This makes optimizing code for accelerators challenging [16, 58], given the large space of candidate code sequences, the fact that providing an accelerator for each compilation run is costly, and that engineering teams often optimize for the next generation of accelerators even before the hardware is available.

Third, when deploying a system, engineers often need *guar-*

antees on performance properties. Consider an autonomous-vehicle driving system that integrates accelerators for real-time image decoding, object detection, and object recognition. To guarantee safe navigation in all operating conditions, engineers must be able to precisely know, for example, the upper bound on image decoding latency. There exists no good way to "verify" the performance of third-party accelerators today. The state of the art is blackbox testing, which is rarely sufficient, so system designers typically rely on heuristics and accumulated wisdom [35]. Given that accelerators are expected to become ubiquitous [63, 79], this status quo must change.

We argue that hardware accelerators should come with standardized *performance interfaces* [42, 43, 45] that summarize performance behavior just like semantic interfaces summarize functionality. Software engineers routinely use semantic interfaces such as code documentation or header files to quickly find answers to questions like what a system call does, or which library is best suited for their requirements, or how incorporating a library will affect their system's functionality as a whole. Since an accelerator's raison d'être is performance (after all, its functionality could come just as well from software running on a general-purpose processor), performance interfaces are as integral to the correct use of accelerators as are semantic interfaces. As explained above, using an accelerator without a performance interface can fail to deliver on the acceleration promise, or even make performance worse.

We propose a new abstraction for representing accelerator performance that makes performance interfaces possible for hardware accelerators; we call this abstraction a *Latency Petri Net* (LPN). An LPN distills only the performance-relevant details of a circuit and excludes all other information, such as functionality. This distillation enables LPNs to serve as a high-fidelity intermediate representation (IR) of a circuit that is *performance-equivalent*: it takes the same inputs as the original circuit, and its performance behavior matches that of the original circuit. The semantics of the LPN circuit's outputs, however, are different. We envision accelerator developers manually producing the LPN as part of their regular design process, and shipping it with the accelerator. We show that doing so is both straightforward for accelerator developers (takes a few hours) and enables them to better understand and debug their own designs. We also show that the LPN of an accelerator need not disclose proprietary intellectual property.

We develop *a toolchain* (ltc) that, based on an accelerator's LPN, automatically produces performance interfaces in the form of simple, human-readable Python programs. Software engineers can use these interfaces to make informed decisions at the system design stage without needing to write code or to purchase the accelerator. ltc also provides a performance simulator that helps engineers understand how to optimize their code. Since the LPN distills only performance-relevant details, ltc's performance-only simulator is orders of magnitude faster than its state-of-the-art cycle-accurate counterparts that also simulate functionality. Finally, ltc also provides a formal verification tool that enables software engineers to prove key performance properties before deploying their systems (e.g., latency bounds for a specific but potentially infinite class of workloads). Our toolchain prototype works well for fixed-function ASICs (e.g., TPU [49] or the accelerators on SoC-based SmartNICs [4, 9, 53]) and simple programmable accelerators. General-purpose programmable accelerators (e.g., GPGPUs) are left for future work.

We demonstrate ltc's effectiveness on accelerators used for deep learning, serialization of RPC messages, JPEG image decoding, genome sequence alignment, and on a Reconfigurable Match Tables (RMT) pipeline used in programmable network switches. We show that the LPN intermediate representation can precisely capture the latency and throughput of various accelerators. Even after LPN simplifications that trade accuracy for simulation performance, we show that the IR still has an average performance-prediction error of only 1.7% across all accelerators. We present a variety of use cases for the resulting performance interfaces and LPNs, including: enabling informed decision-making during the system design stage without requiring elaborate benchmarking; cycle-level performance simulation that is up to 7821× faster than state-of-the-art cycle-accurate simulators, enabling ML compilers to generate code optimized for the accelerator in seconds instead of hours; and using formal verification to gain confidence in an accelerator's performance before deploying it in production.

The rest of this paper is organized as follows: We provide an overview of our proposed solution (§2), then define the new LPN abstraction (§3) and describe the ltc toolchain (§4). We then evaluate ltc experimentally (§5), discuss further ideas (§6), present related work (§7), and conclude (§8).

2 Design Overview

To help software engineers reason precisely about accelerator performance, we introduce the Latency Petri Net (LPN) intermediate representation: an abstraction of the accelerator's implementation that is *performance-equivalent*, i.e., its performance behavior (but not functional output) matches that of the original circuit. We then propose a workflow that uses the LPN to answer key questions about accelerator performance at the system design, implementation, and deployment stage via an extensible toolchain that we call ltc.

The LPN is inspired by classic Petri nets [69], a class of graphs used for the description and analysis of concurrent systems and processes. They are used in various domains, including the design and verification of digital asynchronous circuits. We define the LPN in §3, and Fig. 2 shows an example.

Petri nets are a good starting point for the LPN abstraction, because the key challenge in reasoning about hardware performance is not reasoning about the individual components but rather about the end-to-end performance that emerges when these components (e.g., multiple pipeline stages) operate together, in parallel. Petri nets were designed to model concur-

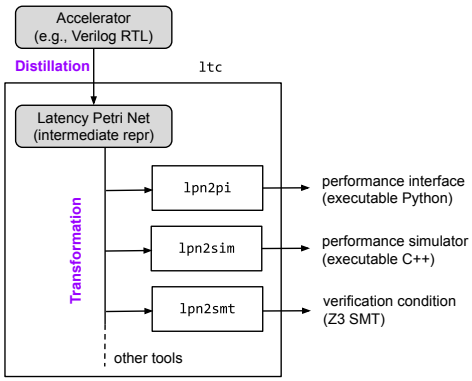

Figure 1: Proposed two-phase workflow: Hardware engineers distill their accelerator design into an LPN IR, and tools transform automatically this IR into the forms desired by accelerator users.

rent systems, and they can precisely capture hardware's inherent parallel and asynchronous execution.

Using a Petri net-like representation also ensures that the LPN is easy for accelerator developers to produce. This is because, when generating an LPN, accelerator developers do not need to reason about the impact of parallel execution on performance, rather they only need to (abstractly) represent the individual components and their local interactions. The ltc toolchain takes the final step to fill in the gaps and turn the LPN into forms that can be consumed by humans.

Fig. 1 illustrates our proposed workflow, consisting of two stages that produce and consume the LPN IR, respectively. The first stage (*distillation*) involves manually translating the accelerator's design into its corresponding LPN (we describe this in §3.3). We propose that distillation be performed by accelerator developers as part of their regular design process, but one could also imagine tools that translate Register-Transfer Level (RTL) designs into LPNs. Since the definitive clock frequency of the circuit is decided in the post-RTL synthesis stage, the LPN abstraction represents execution latency in terms of cycles (i.e., an RTL-level metric), not wall-clock time. The latter is easily calculated once the frequency is known.

The second stage of the workflow (*transformation*) automatically processes an accelerator's LPN into actionable information about accelerator performance. The ltc toolchain consists of several tools: lpn2pi summarizes the performance of the accelerator into human-readable, executable Python programs that enable software engineers to make informed development decisions without purchasing the accelerator or porting their code to it. lpn2sim produces an executable simulator of the LPN that developers and tools can use for fast performance simulation while optimizing their code. lpn2smt translates the LPN together with a user-provided performance property into a verification condition and passes it to the Z3 constraint solver [22] for a proof or refutation of the property.

This two-staged workflow—distilling the accelerator design into a performance IR and then transforming the IR into answers to specific questions about performance—provides

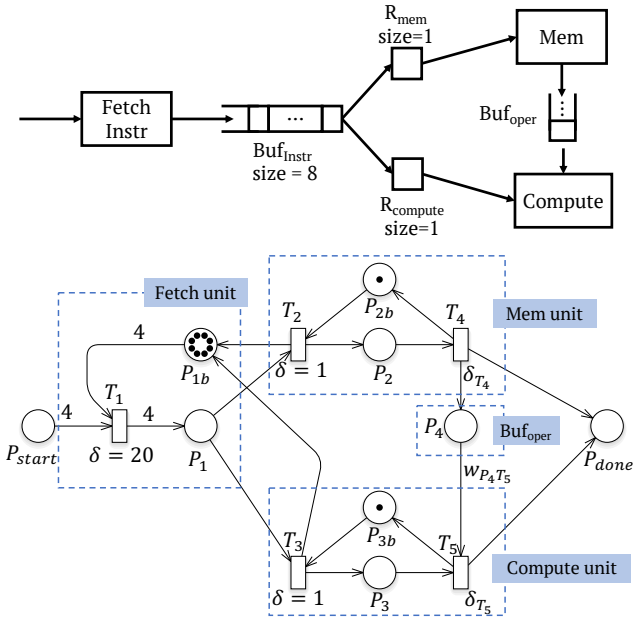

Figure 2: Example hardware pipeline (top) and its LPN (bottom). This is a simplified version of the deep-learning accelerator in §5.

flexibility and customizability. Since the LPN is a universal and accurate representation of the accelerator's performance, it can be transformed into answers to arbitrary questions about the accelerator's performance. We envision the set of tools in ltc expanding over time, to address other questions one might ask about an accelerator's performance.

3 The Latency Petri Net Abstraction

We now define the LPN abstraction, first at a high level (§3.1) and then more precisely (§3.2); a complete formal definition is beyond the scope of this paper. We then describe step-by-step how to distill an accelerator design into its corresponding LPN (§3.3). Finally, we discuss the use of LPNs to model components surrounding accelerators, such as memory and interconnects (§3.4).

3.1 LPN Overview

To illustrate the LPN concept, we use the simple hardware pipeline shown in Fig. 2. It consists of a fetch unit that brings instructions into an instruction buffer, followed by an in-order dispatch to a memory and a compute unit. The memory unit fetches the operands for compute instructions from memory into an operands buffer. The memory and compute units operate in parallel, i.e., the memory unit can fetch operands for a future compute instruction while the compute unit is still processing the current instruction. Each unit operates on one instruction at a time, stored in each unit's local register (R_{mem} respectively $R_{compute}$) until the unit finishes processing it.

The memory and compute units have variable latencies d_{mem} and d_{comp}, respectively, that take into account the in-

struction type and when the operand was last accessed. For simplicity of presentation, we fix the fetch unit's latency to 20 cycles (fetches 4 instructions at a time), set the instruction buffer's maximum size to 8, and let the operands buffer have infinite capacity, unlike in a real accelerator.

Reasoning about the latency of a sequence of instructions is challenging, even for such a simple pipeline, due to the fetch, memory, and compute units operating in parallel. Parallel execution can both hide latencies (e.g., loads that bring in the operands for future compute instructions may complete before the current compute instruction) and introduce stalls (e.g., in the fetch unit due to back pressure when the memory and compute units drain the instruction buffer too slowly).

The bottom half of Fig. 2 illustrates the LPN for this simple pipeline. The LPN is a directed graph with two kinds of vertices: *places* (circles) and *transitions* (rectangles). Adjacent vertices in the LPN must be of different kinds, i.e., edges in the graph can only connect places to transitions and vice-versa. Each place in the LPN contains *tokens* (solid black dots) that collectively represent the state of the circuit, and tokens are stored and consumed in FIFO order.

An LPN models how data flows through a circuit by *enabling* transitions. Each transition has a *guard* (not shown) that determines whether the transition is enabled or not, a *delay* (δ) that specifies the duration of the transition in cycles, and a *producer* function (not shown) that generates new tokens. Once a transition is enabled, after the number of cycles indicated by the delay, it *commits*, i.e., atomically consumes input tokens and produces output tokens. We define each of these operations precisely in the next section.

In Fig. 2, we show the correspondence between the circuit blocks and the subgraphs of the LPN. For some of the LPN details, such as the transition delays, one needs to consult the RTL of the accelerator (not shown). The LPN at the bottom is an abstract representation of a circuit that is performance-equivalent to the one at the top: (1) it operates on the same inputs, using a function (not shown) that converts instructions to tokens in the special place P_{start}; and (2) given any input, the number of cycles it takes the LPN to deposit the last token in P_{done} corresponds to the number of cycles the upper circuit takes to produce its output. However, the LPN's output tokens are meaningless other than indicating completion.

3.2 LPN Definition

In essence, an LPN models a system of queues connected by logic units that consume tokens originating from multiple input queues and generate tokens for designated output queues. The LPN is a directed dataflow graph in which places P_i represent the queues, and transitions T_j represent the logic units. Edges directed from places to a transition are the transition's input edges, while those directed from the transition to places are its output edges. We equip the LPN with a timestamping machinery CLK to denote when each token in the system was produced—this is a key ingredient for modeling performance.

An LPN state $S = ((s_1, \ldots, s_n), t)$ is a tuple consisting of a collection $s_1, \ldots, s_n$ of sequences s_i representing the in-flight tokens corresponding to places $P_1, \ldots, P_n$, and one global non-negative number t, the current value of CLK. A token $k = (p, ts)$ is composed of a map $k.p$ of key-value pairs and a timestamp. Each key in $k.p$ is the name of a property of k. Each token has a type, determined by the set of properties (but not values) that tokens of that type have. All tokens in a particular place have the same type. The timestamp $k.ts$ denotes the CLK value when token k was produced. By construction, the timestamp $k.ts$ of any token in a reachable state S is $k.ts \leq S.ts$. The tokens in a place are always consumed in FIFO order, which is why $s_1, \ldots, s_n$ are sequences and not mere sets.

A transition $T = (\gamma, \delta, \pi)$ is a tuple of three functions: a guard γ, a delay δ, and a producer function π. The guard decides when T is ready to execute: $T.\gamma$ reads (without consuming) a subset of the tokens present in T's input places and returns *NotReady* if the transition cannot execute at this time. If it can, then the guard returns $Enabled(w_1, ..., w_k)$ with weights w_i. To execute, the transition locks the first w_1 free tokens from its 1st input place, the first w_2 free tokens from its 2nd input place, and so on. The guard must guarantee that $\forall i, w_i$ is less than or equal to the number of free (not locked) tokens already present in the transition's i^{th} input place.

When a guard $T.\gamma$ switches from *NotReady* to *Enabled*, thus enabling T, the transition does not immediately consume the tokens but rather locks them for $T.\delta$ cycles. The lock means that no other transition is allowed to consume those tokens. At the end of the delay $T.\delta$, the transition commits: the locked tokens are atomically removed from T's input places, and the tokens produced by $T.\pi$ are pushed to the output places, with the current CLK (commit time) as their timestamp. Both the delay $T.\delta$ and the producer $T.\pi$ are arbitrary functions of all the input tokens that the transition promises to consume.

To avoid race conditions when two transitions share an input or output place, we require that the two transitions never be simultaneously enabled. The value of a guard $T.\gamma$ is not allowed to change between the moment it switches to *Enabled* and the moment when $T.\delta$ has elapsed (and T commits).

LPN Semantics. Given an initial LPN state $S_0 = ((s_1, \ldots, s_n), 0)$ with all the tokens in $s_1, \ldots, s_n$ having a timestamp equal to 0, we define the semantics of the LPN starting from S_0 as the potentially infinite sequence of states inductively defined by $((s_1, \ldots, s_n), t) \rightarrow ((s'_1, \ldots, s'_n), t')$.

The next state of an LPN is obtained by applying the effects of all the transitions that are enabled at CLK $= t$ and known to be ready to commit at CLK $= t'$. For a transition T_i to belong to this group, it must be that the guard $T_i.\gamma$ returned *Enabled* at time $\leq t$, its input tokens locked in the corresponding places were produced before T_i started (i.e., the highest timestamp of those tokens is $ts_{\max} = t' - T_i.\delta$), and the earliest time when T_i can commit is t'. All transitions known to be ready to commit at t' commit as a group, and they advance the LPN from $((s_1, \ldots, s_n), t)$ to $((s'_1, \ldots, s'_n), t')$. When no more transitions

can ever commit, the LPN has reached its terminal state.

In an LPN, it is possible that $T_i.\delta = 0$. If a commit at t' enables such a 0-delay transition, T_i will also commit at t', even if this was not previously apparent, i.e., T_i was not previously "known" to be ready to commit at t'. Thus, after the first commit, the LPN transitions $((s_1,\ldots,s_n),t) \rightarrow ((s'_1,\ldots,s'_n),t')$, and subsequently it transitions $((s'_1,\ldots,s'_n),t') \rightarrow ((s''_1,\ldots,s''_n),t')$, i.e., there are multiple states with the same timestamp. The process repeats until no more transitions can commit at t'.

LPNs are reminiscent of several extensions of Petri nets [46, 69, 86], mixing the notion of timestamp and information-carrying tokens with enforced FIFO ordering between tokens. With an LPN, we can accommodate the different modeling needs of hardware accelerators, while keeping the underlying models formal and machine-analyzable. We designed the LPN to provide a favorable trade-off between compactness, analyzability, expressivity, and ease of manipulation for our different uses and tools. We chose the name "latency Petri net" to acknowledge the inspiration we drew from Petri nets, without implying a theoretical equivalence.

3.3 Distillation: From RTL to LPN

We now describe how a hardware engineer can represent the performance of an accelerator using an LPN.

Distilling an accelerator's register-transfer level (RTL) representation into its corresponding LPN is an element-wise, structural conversion of the RTL: FIFO buffers in the RTL become LPN places, and RTL compute elements become LPN transitions. Transitions can operate in parallel (if enabled at the same time), so the engineer can produce the performance-equivalent representation by analyzing in isolation the latency of each stage of the accelerator pipeline. The LPN then glues back together this stage-by-stage performance decomposition.

RTL-to-LPN distillation is a five-step process; we describe each step in reference to the example in Fig. 2.

Step 1 involves listing the places and transitions that map directly to elements in the RTL: places P_1, P_2, P_3, P_4 correspond to the four buffers/registers, transitions T_1 to T_5 correspond to the three units that consume/produce from/to those buffers plus the two copy actions of instructions to the registers $\mathrm{R}_{\mathrm{mem}}$ and $\mathrm{R}_{\mathrm{compute}}$. The latter two are not explicit computations in the block diagram but are units in the RTL source code.

Step 2 involves defining the guard functions and the corresponding weights. For many transitions, becoming *Enabled* simply requires the presence of a specific number of tokens in an input place; their guards do not look at the properties of those tokens (e.g., $T_4.\gamma$ and $T_5.\gamma$). Occasionally, guards may depend on the values of token properties: $T_2.\gamma$ (respectively $T_3.\gamma$) will be *Enabled* if and only if the first free token in P_1 has a value corresponding to a memory (respectively compute) instruction, because instructions are dispatched in order.

Most weights returned by the guards are constants (e.g., $w_{T_1P_1} = 4$ because the fetch unit fetches 4 instructions at a time, and $w_{P_1T_2} = w_{P_1T_3} = 1$ because both registers store 1 instruction at a time). Default weights of 1 are not shown in Fig. 2. Occasionally, weights may depend on the values of token properties: $w_{P_4T_5}$ determines the number of operands transition T_5 reads, and it is a function of the value of the property of the token in P_3 that specifies the type of instruction. In both cases, the weights are intuitive for accelerator developers to define, because they directly correspond to an architectural quantity: the rate of consumption of tokens in the dataflow. This also illustrates why weights need to be computable based on tokens from all of a transition's input places.

Step 3 involves defining the delay and producer functions for each transition. The delay typically comes straight from the RTL. The producer function produces tokens with just those property values that are strictly necessary for the LPN to accurately model performance—performance-irrelevant should be discarded.

Step 4 involves modeling backpressure by adding capacity constraints to each place in the LPN. Take for example the *Mem* unit: we add an extra "capacity place" (P_{2b}) with a fixed initial number of "capacity tokens", corresponding to the capacity C of the buffer in question (1 token for P_2); this is a classic Petri net pattern [46]. The capacity place is connected to the transitions incident on the original place (T_2 and T_4), but in reverse, to form a loop. We adjust T_4's producer function to also produce 1 capacity token into the capacity place P_{2b}, and T_2's guard to require that there be at least 1 capacity token in P_{2b} to enable T_2. This way, when T_2 first commits and consumes the initial token in P_{2b}, it cannot commit again until T_4 has committed and deposited a capacity token in P_{2b}. This models the *Mem* unit backpressure: no new instruction will be copied into $\mathrm{R}_{\mathrm{mem}}$ until the previous memory instruction has finished processing. The same pattern is applied, for instance, to the P_1 place representing the instruction buffer ($\mathrm{Buf}_{\mathrm{instr}}$), except that there are two consumers for $\mathrm{Buf}_{\mathrm{instr}}$ and the capacity is $C = 8$, thus 8 initial tokens in P_{1b}.

Finally, step 5 involves adding *start* and *done* places (P_{start} and P_{done}), and placing the initial tokens. The hardware engineer then provides a "tokens from input" function Ψ to translate the accelerator's input to the LPN tokens placed in P_{start}. A stream of input data (e.g., an image) can be split into task units (e.g., individual blocks), and each task becomes a token that is placed inside P_{start}. Depending on the accelerator's semantics, a task token could also be an instruction, a short DNA sequence, etc. When the processing of a task completes, a "done token" k_{done} should be produced into P_{done}.

Constructing LPNs is a natural fit for accelerator development workflows and a materialization of what hardware engineers already have in mind, i.e., a more detailed architectural diagram annotated with latency expressions. Compared, for instance, to building a simulator, producing an LPN is easier, because the accelerator functionality is abstracted away. ltc provides a Python library with built-in types for places, transitions, edges, etc. that engineers can use to write the LPN.

We asked a hardware engineer to produce an LPN for the Menshen RMT pipeline used in programmable network switches [82]. After taking 3 days to understand the RTL design, he produced the corresponding LPN (which we evaluate in §5) in less than 3 hours. This suggests that the manual distillation step is indeed straightforward for someone who understands the accelerator's design. The same engineer also mentioned that writing the LPN actually helped to better understand the performance behavior of the circuit.

Finally, in most cases, LPNs do not leak much proprietary information about the accelerators. Except for the latency details, an LPN reveals no more information than the high-level architectural diagrams, which are often made public anyway. No implementation details appear in the LPN.

3.4 Memory, Caches, and Interconnects

Accelerators are often part of a larger system, and their performance is influenced by the components surrounding them, such as memory, caches, and interconnects. LPNs can be used to model these components as well. However, they provide fewer benefits over other kinds of performance models than they do for accelerators.

First, LPNs can be constructed even without a reference RTL implementation, by speculatively modeling the internals of a hardware component based, for instance, on documentation and online posts. We built an LPN for a sophisticated PCIe interconnect based on documentation alone, and we describe this example in §5.

Second, modeling complex memory hierarchies is challenging, because semantics are tightly intertwined with performance: the latency of a cache access depends on which entries are present in the cache or not, and knowing this requires tracking the specific contents of the cache, which in turn requires modeling the semantics of the cache in more detail than for most accelerators. This is an example where the ability of an LPN to abstract away functionality is limited, and thus its advantage over, say, a cycle-accurate simulator is reduced. One could model the state of the entire cache with a single token, and each cache line would be an individual property of that token. This LPN, though, would likely be more complex than what the ltc toolchain was designed for.

Nevertheless, an LPN can still abstract away some semantic details of the memory hierarchy and be productively used, for instance, to model and reason about the parallelism within the memory subsystem. If we took the RTL of a cache and distilled it into an LPN by following the steps discussed in §3.3, we could model the cache's internal logic (without taking into account cache state) and simulate it with lpn2sim. This could help reveal that a particular cache design can only handle 1 cache hit every 2 cycles, whereas a better design could handle a cache hit every cycle, through pipelining. The pipeline design does influence cache performance, even if not as much as replacement strategy and associativity configuration do. An LPN can help fine-tune the pipeline design.

4 Transforming the LPN

The LPN is an abstraction that is *performance-equivalent* to the accelerator; nevertheless, it is not easy to read for those unfamiliar with the accelerator's implementation details. As a result, it is not directly useful to software developers who want to use the accelerator in their systems. We now describe how the ltc toolchain bridges this hardware–software gap by transforming the LPN into representations that software developers can use in the different stages of system development.

The ltc toolchain currently consists of three main tools: (1) lpn2pi, which transforms the LPN into human-readable performance interfaces in the form of executable Python programs, which are meant to be read as much as executed; (2) lpn2sim, which merges the LPN with a simulator skeleton to produce an executable simulator that both developers and tools can use for fast performance simulation; and (3) lpn2smt, which translates the LPN together with a user-provided performance property into verification conditions that can be proven or refuted using an SMT solver, to provide performance guarantees before the system is deployed. ltc also provides other, simpler tools that we do not describe here, such as lpnviz, which produces a visualization of the LPN that hardware developers can use to better understand and debug the accelerator. We envision both hardware and software engineers contributing more such tools to ltc over time.

While the lpn2sim simulator (just like the RTL) operates on *concrete* inputs, both lpn2pi and lpn2smt produce outputs—performance interfaces and verification conditions, respectively—that describe performance for an *abstract, symbolic* input. Since the space of all possible inputs to an accelerator is large, often infinite, producing complete performance interfaces or verification conditions is intractable for most LPNs, due to the path explosion problem [11]. We circumvent this challenge by introducing the notion of *input classes* for LPNs, which partition a given input space into input sets for which, individually, it is feasible to produce complete performance interfaces and verification conditions. We now describe how input classes partition an input space (§4.1), and then describe lpn2pi (§4.2), lpn2sim (§4.3), and lpn2smt (§4.4).

4.1 Input Classes for lpn2pi and lpn2smt

To use the lpn2pi and lpn2smt tools, one must first constrain the input space to the one of interest. For example, for the JPEG Decoder, the user might include all images up to a given maximum size (number of pixels $\times$ pixel depth in bits) and exclude all others. This input space is then partitioned by an ltc tool into *input classes*, with lpn2pi and lpn2smt then solving the problem for each class independently. This ltc tool employs symbolic execution [13] to partition the input space.

Intuitively, an *input class* is a group of inputs for which simulating the LPN will cause (1) each transition in the LPN to commit exactly the same number of times for all executions corresponding to inputs in that class; and (2) the n^{th} commit

of each transition will consume and produce the same number of tokens in all executions, for all values of n. (An "execution" is a complete simulation of the LPN from CLK=0 until it deposits the last k_{done} token into P_{done}.) For example, if executing the LPN with an input from a class causes transition T to commit twice during the execution, consuming 3 tokens for the 1st commit and 4 tokens for the 2nd commit, then executing the LPN with any other input from that class must also cause T to commit twice and to consume 3 tokens for the 1st commit and 4 tokens for the 2nd commit. The tokens consumed and produced in different executions must have the same type (§3.2), but can have different property values. Commits of different transitions can be interleaved arbitrarily in different executions for inputs in a class; the only thing that matters is the commit and token counts. §A.1 in the Appendix contains a formal definition of input classes.

Input classes are defined such that all inputs in any given class impose the same pattern on the trace resulting from the simulation of the LPN. The tools leverage this commonality to do their analysis once per pattern (which could subsume many inputs). Even though input classes are not defined based on human-understandable semantics of the accelerator's input, they often do correspond to input types that are intuitive for users. For example, for the Protoacc LPN (§5), all messages of a given format constitute one input class. For the JPEG decoder LPN (§5), all images of the same size form a separate input class.

As mentioned, ltc includes a preprocessing tool for automatically partitioning the user-specified input space into input classes. It symbolically executes the LPN in a special way and partitions the input space into sets. One input class can possibly span multiple sets, but a set never contains inputs from more than one input class. Then, by operating on each set in isolation, lpn2pi and lpn2smt can avoid path explosion and are trivially parallelizable by input set. Please see A.2 in the Appendix for details of how input classes are generated.

4.2 lpn2pi

The lpn2pi tool transforms the LPN into human-readable performance interfaces represented as executable Python programs, in the spirit of [42, 43]. The performance interface takes the same inputs as the accelerator (e.g., a stream of network packets, multiple RPC messages, a long DNA sequence) and returns the start-to-end latency (i.e., total execution cycles) it would take the accelerator to process that input. The performance interface describes the start-to-end latency not with concrete numbers but with *formulae*, as introduced in [44] but expressed in terms of properties $k.p$ of initial tokens k in the accelerator's LPN. The performance interface returns a single formula per input class. We show examples of lpn2pi-extracted performance interfaces in Figs. 4 - 7.

lpn2pi does not aim for fully precise performance interfaces—while the LPN has suitable constructs to precisely represent the accelerator's asynchrony and parallelism, reflecting these in a precise, closed-form, human-readable formula is typically intractable. Instead, lpn2pi *approximates* the accelerator's start-to-end latency, trading precision for human readability and closed-form expressions. Approximation turns out to be sufficient, because we expect performance interfaces to be used mostly during the system design stage, when software engineers are interested in coarser-grained descriptions of performance. In §5, we show that, while approximate, lpn2pi-extracted interfaces nevertheless enable informed decisions at the design stage, such as choosing the accelerator configuration that fits best a particular workload profile.

The following three assumptions underlie the approximation made by lpn2pi: (1) The size of the input is large enough so that the accelerator's pipeline is almost always full, i.e., the time spent filling and draining the pipeline is a negligible fraction of the overall start-to-end latency; (2) For all inputs in any given input class, the accelerator has the same bottleneck, i.e., the stage in the accelerator's pipeline (or transition in the LPN) that incurs the longest delay is the same for all inputs in that class; and (3) The bottleneck in the accelerator pipeline is stable, i.e., it does not shift from one pipeline stage to another during the processing of an input.

We define the effective delay ε_T of a transition T in an execution of an LPN as the product $N \times \overline{g}_T$ of the number of times N the transition commits in that execution and the average duration between consecutive commits of that transition (called average commit gap $\overline{g}_T$). The transition with the largest effective delay is deemed to be the bottleneck. Based on the three assumptions above, lpn2pi approximates the start-to-end latency to be the effective delay ε_T of the bottleneck transition (i.e., of the bottleneck stage of the pipeline). Recall that the definition of an input class (§4.1) requires the number of times each transition commits in an execution to be the same for all inputs in that class. So, given an input class and a transition T, N is a constant for all inputs in that class. However, $\overline{g}_T$ is a symbolic expression parameterized by properties of the initial tokens, so ε_T is also a symbolic expression.

Determining ε_T for each transition comes down to determining the respective $\overline{g}_T$. Note that $\overline{g}_T$ does not necessarily equal $T.\delta$, because a transition may be stalled for an arbitrary number of cycles (due to its input places not having enough tokens) or it could be non-blocking and become enabled again before it commits (i.e., commit multiple times in parallel).

Accurately estimating the average commit gap $\overline{g}_T$ is challenging in LPNs with loops. In a loop-free LPN, $\overline{g}_T$ for all transitions is just the maximum of all transition delays. But with loops, this simple method is no longer accurate. Consider a simple loop $P_0 \to T_1 \to P_1 \to T_2 \to P_0$, with initially a single token in P_0 and none in P_1. Every time T_1 commits, it subsequently has to wait for T_2 to commit before it can be enabled again (and T_2 also has to wait for T_1), so the average commit gap for both T_1 and T_2 is the sum of the delays of T_1 and T_2.

To approximate the $\overline{g}_{T_i}$ of all transitions T_i in a loop, we define the loop delay Δ and a parallel factor F_{T_i} for each tran-

sition T_i in the loop—we estimate $\overline{g}_{T_i}$ as Δ/F_{T_i}. Here is why: Consider a simple loop that has only one place with M initial tokens. Δ is the time it takes for the M initial tokens to complete a full iteration around the loop, with all transitions committing at least once. Recall that transitions are non-blocking, in the sense that a transition T_i, once enabled, could become enabled again before it commits, if the configuration of free tokens in its input places changes such that $T_i.\gamma$ is satisfied once again. As a result, there can be multiple instances of the same transition enabled at the same time. By F_{T_i} we denote the maximum number of instances of T_i that can be simultaneously enabled. Increasing levels of concurrency proportionally reduce the gap between successive commits, so we estimate the average commit gap for T_i as $\overline{g}_{T_i} = \Delta/F_{T_i}$.

lpn2pi starts out by setting $\overline{g}_T = T.\delta$ for each transition T in the LPN; if $T.\delta$ is a general function on input tokens, $\overline{g}_T$ is the corresponding symbolic expression. Then, for each loop in the LPN, lpn2pi computes Δ and the F_{T_i} factors, and then it recomputes $\overline{g}_{T_i} = \max(\overline{g}_{T_i}, \Delta/F_{T_i})$ for each T_i in that loop. Once lpn2pi has treated each loop once, we say that it has completed one iteration of the process. Dependencies among loops and the order in which the loops are treated can influence the estimated $\overline{g}_{T_i}$ values, so lpn2pi continues iterating in order to improve the estimates. The current version of lpn2pi stops after a fixed number of iterations that is configurable (default 10). In the next version, lpn2pi will automatically stop when the $\overline{g}_{T_i}$ values stabilize, i.e., do not change from one iteration to the next by more than a configurable threshold. Comparing changes at the level of symbolic expressions (which is what some $\overline{g}_T$ values are) is fundamentally hard, so lpn2pi will instead compare concrete values of these expressions, obtained by randomly sampling the input class and computing the corresponding concrete values of the expressions. Once iterative estimation is complete, lpn2pi derives the start-to-end latency formula as the maximum of the ε_T expressions (i.e., $max_{T_i}(N \times \overline{g}_{T_i})$. lpn2pi then uses `sympy` [61] to simplify the formula to obtain an expression that is easier for humans to read.

lpn2pi repeats the process described in the previous paragraph for each input class, after which it emits the corresponding interface program in Python, in the form seen in Figs. 4 - 7.

Please refer to A.3 in the Appendix for more details on the computation of ε_T, Δ, F_{T_i} and on the underlying assumptions.

4.3 lpn2sim

For a given LPN, the lpn2sim tool produces a bespoke cycle-level performance simulator that can be used by both engineers and tools. As we show in §5.3, the LPN's power of abstraction enables lpn2sim-generated simulators to simulate performance orders-of-magnitude faster than state-of-the-art cycle-accurate simulators.

lpn2sim's simulator is event-driven and works as follows: In step ⓪, it sets the value of CLK to zero, and then repeatedly performs the following two steps to make forward progress. In step ①, the simulator finds all transitions that can commit at the current CLK value. If more than one can commit, the one with the smallest ID is committed first. The simulator repeats this step until no transition can commit at the current CLK value. In step ②, the simulator finds the next earliest timestamp at which a transition can commit. If no transition can commit, the simulation terminates. Else, the simulator updates the CLK to that timestamp and goes back to step ①.

lpn2sim automatically translates the LPN (described by hardware engineers using our Python API) to an equivalent C++ program. It first emits equivalent place and transition objects in C++, and then translates individual delay, guard, and output functions. To make such automatic translation of delay, guard, and output functions feasible, our Python API only allows arithmetic operations and conditionals within these functions; this proved sufficient for all the accelerators we evaluated. lpn2sim then combines the translated LPN code with a simulator skeleton we wrote in C++ and compiles to an executable.

4.4 lpn2smt

The lpn2smt tool is used to formally reason about performance properties of an accelerator based on its LPN. It has three inputs: the target LPN, the input space Υ, and a query Φ. The Υ parameter is the subspace of inputs that are of interest to the user. lpn2smt currently supports queries related to latency bounds of two kinds: what is the upper (lower) bound on latency, or can you prove/disprove expression $\phi(x)$ involving latency x. An example of the latter, which we use in our evaluation, is $\phi(x) : |(E-x)/x| < 0.2$, where E is an expression taken from an (approximate) performance interface. This asks for a formal proof that E is within 20% of the true latency for all possible inputs and, if not, asks for a counter-example. lpn2smt can be extended to support other kinds of queries too.

lpn2smt first partitions Υ into input classes (§4.1), then derives based on the LPN a precise SMT expression Λ_i for the start-to-end latency for each input class i. Then it pieces together a global latency expression for the entire input subspace Υ as $\Lambda_\Upsilon = \mathsf{ite}(C_1, \Lambda_1, \mathsf{ite}(C_2, \Lambda_2, \mathsf{ite}(...)))$ using the if-then-else ite operator supported by SMT solvers like Z3 [22] and the inputs constraints C_i that define the corresponding input classes. Note that this is a precise expression, not like lpn2pi's approximations meant to be human-readable.

For the first kind of query, lpn2smt passes Λ_Υ to the SMT solver's optimizer and asks for a formally verified upper (lower) bound on Λ_Υ. For the second kind of query, lpn2smt passes $\phi(\Lambda_\Upsilon)$ to the SMT solver and returns either a formal confirmation that it is true or a counter-example.

The SMT expression Λ_i for input class i is constructed as follows: Let N_i be the number of transition commits in an execution from this class; by the definition in §4.1, there is a unique N_i for each input class i. lpn2smt instantiates N_i symbolic timestamps CLK_1, CLK_2, ... corresponding to when the transitions committed—the start-to-end latency will be $max_{j=1..N_i}\{\text{CLK}_j\}$. To compute this expression, lpn2smt in-

stantiates for each commit j the consumed tokens, and places them in the corresponding input places. For each such token k, lpn2smt sets $k.p$ and $k.ts$ to symbolic values constrained to reflect the relationship to CLK_j. Recall that the number of tokens produced/consumed by a transition is the same for all inputs in a class (§4.1). Then lpn2smt uses the "tokens from input" function Ψ (applied to a suitably constrained symbolic input from this class) to obtain the initial tokens, and places them in P_{start}. It then uses the transitions' producer functions to instantiate the transition-produced tokens into the corresponding places. For each initial and produced token k, lpn2smt constrains $k.p$ and $k.ts$ according to the function that produced it and the respective $T.\delta$ and CLK_j. Then, lpn2smt captures the constraints resulting from the fact that every consumed token must either be an initial token or one resulting from a commit. The constraints that result are propagated to CLK_1, CLK_2, ..., and finally lpn2smt computes $\Lambda_i = max_{j=1..N_i}\{\text{CLK}_j\}$.

In summary, the LPN is a generic IR that is performance-equivalent to the accelerator circuit and can be transformed by tools into higher level representations useful to software engineers. In this section, we presented three of the tools in the ltc toolchain: lpn2pi, lpn2sim, and lpn2smt. We envision both hardware and software engineers contributing more such tools to ltc, increasing its usefulness over time.

5 Evaluation

In this section, we evaluate ltc on several accelerators and show that it answers the questions mentioned in §1. We first describe our experimental setup (§5.1), then present fine-grained results that shed light on detailed aspects of LPNs (§5.2), and conclude with higher-level results (§5.3).

5.1 Experimental setup

We evaluate ltc on 5 accelerators (Table 1), each representative of a particular class of accelerators. We require access to the RTL, so the evaluation is limited to open-source accelerators.

Apache VTA (Versatile Tensor Architecture) [2] is a deep-learning accelerator with a compiler stack based on TVM [15]. The accelerator incorporates tensor cores that perform vector or matrix operations. The design includes parallel units for compute, load and store operations, which decouples memory accesses from the compute, to hide memory latencies [74]. VTA can be used to program arbitrary dataflows when executing the deep-learning model. Certain high-level machine learning operations can be implemented with different VTA instruction sequences. Each instruction sequence exhibits different performance, and so TVM (VTA's compiler) generates multiple instruction sequences and selects the best performing one. This process is called auto-tuning. Our evaluation uses a workload consisting of 1,500 instruction sequences generated from auto-tuning ten 2d convolution tasks from ResNet-18 [34], an 18-layer deep convolutional neural network commonly used to measure auto-tune latency and inference speed.

Accelerator	Domain	Workload	LOC
VTA [2]	Deep learning	Autotune ResNet-18 [34]	6,628 Chisel
Protoacc [53]	RPC message serialization	Hyperprotobench [31] and microbenchmarks	3,197 Chisel
JPEG [80]	Image decoding	30K Flickr [51] and 30K Div2k [50]	7,003 Verilog
Darwin [20]	Bioinformatics	10 DNA test sequences [21]	1,535 Verilog
Menshen [82]	Programmable P4 switch	3 Verilog testbenches (with up to 100 packets)	11,169 Verilog + 4,318 VHDL

Table 1: Open-source accelerators used for evaluating ltc.

Protoacc [53] is a hardware accelerator developed by Google for protocol buffers [71] and integrated into a RISC-V SoC. We only consider Protoacc's serializer, which is the most interesting part of Protoacc: multiple fields within a message are serialized in parallel within the accelerator. Deserialization is sequential and thus less interesting. As in the evaluation of the ProtoAcc paper [53], we use the Hyperprotobench benchmark [31] and their microbenchmarks to measure serialization performance of both large messages (>1MB) and small messages (<1KB). While Protoacc's standard testbench includes a complex memory subsystem (with caches, DRAM, and TLB), we are only interested here in the performance of the accelerator itself, i.e., what a vendor would provide an LPN for. Therefore, in the empirical measurements, we warm up and overprovision the caches and TLB to prevent them from disturbing the performance of the accelerator.

JPEG [80] is an image decoder core for FPGAs written in Verilog. It supports various chroma, fixed and dynamic Huffman tables, DQT tables for JPEG input streams, etc. Our workload consists of the Flickr [51] and Div2k [50] datasets. Each has 30K diverse images, and all images in the Div2k dataset are high-resolution.

Darwin [20] is a GACT (DNA sequence) alignment accelerator. The accelerator has two main stages. The first stage uses a systolic array to fill scores in a 2D score matrix, and the second stage computes alignment actions at each step: insertion, deletion, and match. For the workload, we use ten pairs of test DNA sequences used by the Darwin authors [21].

Menshen [82] is a Reconfigurable Match Tables (RMT) pipeline used in a programmable P4 network switch [12]: incoming packets are processed by flowing through a programmable packet filter, 2 packet header parsers, 5 header processing stages, and 4 header de-parsers. Menshen extends the RMT architecture with isolation mechanisms to ensure that multiple P4 programs running on the same switch do not suffer from performance interference. It spatially partitions its stateful resources (match-action table entries and stateful memories) and uses per-packet configuration overlays for its stateless resources (packet filter, header parsers, header processing stages, and header de-parsers). As workloads, we use Menshen's two original device-level testbenches, plus an additional testbench based on the original but extended to 100 packets. Menshen contains several closed-sourced IP blocks,

which restricts some of our experiments.

We ran all experiments on a 2-socket 48-core Intel Xeon Gold 6248R processor with 376 GiB of memory, 1 thread per core, running Ubuntu 20.04.4 LTS with the 5.15 Linux kernel. For the speedup and accuracy baselines, we compare to Verilator [81], the fastest open-source cycle-accurate RTL simulator available today—it generates optimized C++ code from Verilog that is 200–1000× faster than interpreted simulators [81]. We use Verilator v5.010 for all accelerators except for VTA, where we use v4.022, for compatibility reasons. All speedup comparisons are single-threaded. Verilator v4.022 and v5.010 have negligible performance differences on a single thread. We use the Clang-11.1.0 compiler. For the PCIe experiments, we use an AMD Alveo U200 accelerator card connected with a gen3 x16 PCIe interconnect to a host without DDIO.

We build the LPNs for the above accelerators by manually inspecting the RTL source code. The LPNs use tokens to abstractly represent the data of various formats and units that flow through the real hardware. For example, input packets in Menshen are turned into tokens with a property representing the type and length of a packet, each 8×8 image block in JPEG is turned into a token with a property representing the number of non-zero pixels after quantization, each instruction in VTA is turned into a token with properties representing different parts of the decoded instruction, and each field in a message in Protoacc is turned into a token with properties representing the type of the field and field length. LPN transitions represent the different hardware components that operate in parallel, and LPN places represent the buffers.

5.2 Understanding LPNs in detail

We now provide a quantitative deep-dive into the LPN abstraction, and we also describe how hardware engineers can themselves use LPNs to better understand and debug their designs.

5.2.1 Accuracy and completeness of the LPN

As explained in §2, the LPN representation enables accelerator developers to describe performance in terms that are familiar to them, and then rely on the ltc toolchain to translate the LPN to representations palatable to software engineers.

Fig. 3 shows that using the LPN as an IR is justified: across all benchmarks and all accelerators, the average latency prediction error of the simulator generated by lpn2sim based on the LPN is 1.7%. The maximum error never exceeds 10%. For the LPN to be 100% accurate, it would need to retain almost all the RTL-level details, which is unnecessary in practice.

This means that the LPN provides a performance IR that is highly accurate and complete, i.e., it contains all the necessary details to provide predictions that are close to reality. Tools based on the LPN IR can therefore achieve high accuracy.

5.2.2 Representation efficiency

Besides accuracy and completeness, the utility of an LPN also depends on its conciseness, ease of update, understandability by non-technical staff, and so on. As we will show in Fig. 8, by incorporating only performance-related details and nothing else, the LPN brings about orders-of-magnitude improvements in simulation time. This is one measure of representation efficiency. In Table 2 we show the complexity of the LPNs along different dimensions, which serves as another measure of representation efficiency.

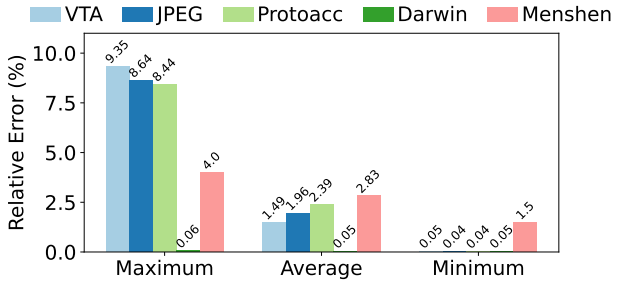

Figure 3: Relative latency prediction errors of the LPN-based simulation vs. Verilator cycle-accurate simulation.

Accelerator	LOC		Number of ...		
	RTL	LPN	transitions	places	edges
VTA	6628 Chisel	506	12	22	41
JPEG	7003 Verilog	109	6	16	33
Protoacc	3197 Chisel	758	97	112	365
Darwin	1535 Verilog	214	2	4	6
Menshen	11169 Verilog	544	29	44	85

Table 2: Comparative complexity of LPN and RTL representations.

5.2.3 Hardware engineer effort to write LPNs

As already mentioned, we asked an accelerator developer with several years of mixed academic and industry experience to read §3 and write an LPN for Menshen, whose design he had not seen before. He wrote the LPN without assistance, and then tested its accuracy with the Menshen testbenches. After understanding the RTL design, it took him less than 3 hours to write an accurate LPN. He estimated that a developer who knew the design and did not need to go back and forth between the RTL and the LPN would take less time.

The Menshen code base is quite substantial. This result therefore strongly suggests that hardware engineers would find it acceptable and practical to write LPNs for their accelerators, especially if they stand to gain (as we argue below).

5.2.4 Utility to SoC and accelerator developers

Besides being easy to write, we believe LPNs, accompanied by the ltc toolchain, can improve the productivity of accelerator designers. For example, finding the right configuration (e.g., sizing the buffers in a programmable switch) is today labor-intensive and error-prone. The wrong choices for buffer sizes can affect the delicate internal balance of an accelerator and lead to performance degradation due to unnecessary stalls.

We discovered the utility of lpn2smt in optimizing buffer sizes while trying to prove an upper bound on the stall-to-cycles ratio (≤ 0.4) for JPEG. In the default configuration, due to an under-sized buffer in the output unit, the previous unit was backpressured early. We had lpn2smt find a buffer size that respects the desired stall-to-cycle ratio: we made the buffer size symbolic and queried lpn2smt to optimize the stall-to-cycles ratio. lpn2smt took 3 minutes to return 0.276 as the optimal ratio and a concrete buffer size that satisfies that ratio. Changing the buffer size (1 line of RTL) led to a 37% performance improvement on the Div2k dataset [50]. Since only some of the images have a stall-to-cycle ratio > 0.4, such a nuanced performance bottleneck would be hard to find.

5.2.5 LPNs beyond accelerators

The performance of the interconnect and external memory accesses affects overall performance when running accelerators, so engineers may want to connect LPNs for accelerators to models for the interconnect and memory, to understand the overall system performance. LPNs are a natural fit for modeling interconnects. We inferred hardware details from PCIe documentations [66], then created an LPN for a reconfigurable PCIe topology, including root complex and switches, and connected it to the LPN for JPEG. With a fixed memory-access latency model, the LPN-based system model achieves on average 1.9% (maximum 5.1%) relative error compared to the end-to-end latency measured with the real hardware system (i.e., a JPEG decoder on an FPGA connected to the host CPU via PCIe). The image set we evaluated on includes 40 images of varying sizes, and the per-image latency ranges from 15 microseconds to 100 milliseconds.

5.3 Key results

In this section, we present high-level results that illustrate the value of LPNs and ltc to accelerator developers and users.

5.3.1 Performance interfaces are human-friendly

This set of results illustrate how LPNs and ltc can answer questions like "What latency/throughput can I expect from this accelerator for my code?" and "Which of accelerators X or Y will best accelerate my workload?". For the latter, we were unable to find two open-source accelerators that provide identical functionality, and so we demonstrate this use case using two configurations of the same accelerator.

Consider the JPEG performance interface in Fig. 4, produced by lpn2pi (as explained in §4.2, the variable names come from the token property names). A quick read conveys that the latency of decoding an image grows with the number of blocks in the image; the compression ratio, which is inversely related to the number of non-zero elements in the block, affects the latency as well. Developers can visually infer the bounds on accelerator latency. To derive a latency in seconds, one multiplies the cycles by the clock period. The `@perf_interface` decorator adapts the input, based on the "tokens from input" function Ψ (§3.3), to make the token properties (e.g., `num_blocks` and `avg_num_nonzero_perblock`) available to the interface at the right level of abstraction.

```
freq = 75*10**6 # 75MHz
clk_period = 1/freq
@perf_interface
def latency_jpeg_decode(img):
    x = 6*(img.avg_num_nonzero_perblock*3+6)
    cycles = img.num_blocks*max(x,509)/4
    return cycles*clk_period

@perf_interface
def tput_jpeg_decode(img):
    # Images are processed one-by-one
    # We provide throughput for RGB blocks instead
   return img.num_blocks / latency_jpeg_decode(img)
```

Figure 4: Latency and throughput interfaces for the JPEG decoder. Comments are manually added. The throughput interface is manually constructed based on the latency interface.

If developers understand the parameters of their workloads, they can directly look at the performance interface to reason about the latency distribution for those workloads. Otherwise, they can generate test cases and quickly run them with the performance interface, which is executable Python code.

Next, consider the performance interface for Protoacc (Fig. 5), which directly conveys the cost of serializing different message types. The latency for serializing a series of messages is just the sum of the latency of serializing individual messages. As mentioned earlier, lpn2pi extracts the performance interface for each input class—in this case, input classes correspond to Protoacc message types—and assembles them together. The performance interface raises an error if the input message is not part of the input classes for which the performance interface was extracted. Due to space limitations, we do not show throughput interfaces, as they are straightforward to derive from the latency interface.

We use two configurations of Protoacc to demonstrate how performance interfaces can help developers choose between accelerators, or between different configurations of the same accelerator, by comparing their performance interfaces (Fig. 5). The first configuration is the original Protoacc, and the second is a smaller configuration of Protoacc with the number of parallel serialization pipelines reduced from six to one. From the interface, if the message type is *hpbench.m1*, we can infer that, if the total bytes are below 47KB, the original Protoacc is faster. And once the total bytes exceed 47KB, the alternative configuration is faster. This is because, when the message size is below 47KB, the bottleneck is still in processing the message—since the original Protoacc has more pipelines to process the message in parallel, it is faster. Once the message size exceeds 47KB, the bottleneck shifts to the generation of the memory reads/writes, and the alternative configuration is faster, because it has a higher frequency.

Finally, Fig. 6 shows the extracted performance interface for Darwin, and Fig. 7 shows the performance interface for

```
freq = 1.8*10**9 # 1.8GHz
clk_period = 1/freq
@perf_interface
def latency_protoacc_serialize(msgs):
    cycles = 0
    # Iterate over each message of a list of messages
    for msg in msgs:
        # hpbench.m* are Hyperprotobench msg formats
        if msg.type == hpbench.m1:
            cycles += max(1468, msg.total_bytes/16+310)
        elif msg.type == hpbench.m2:
            cycles += max(2172, msg.total_bytes/16+514)
        elif msg.type == hpbench.m3:
            ...
        else:
            raise NotImplementedError(
                f"message type not supported"
            )
    return cycles*clk_period
```

```
freq = 2*10**9 # 2GHz
clk_period = 1/freq
@perf_interface
def latency_protoacc_alternative_config_serialize(msgs):
    # Latency interface for another protoacc
     configuration where the number of parallel
     pipelines is reduced from 6 to 1.
    cycles = 0
    for msg in msgs:
        if msg.type == hpbench.m1:
            cycles += max(3609, msg.total_bytes/16+310)
        elif msg.type == hpbench.m2:
            cycles += max(4566, msg.total_bytes/16+514)
        elif msg.type == hpbench.m3:
            ...
        else:
            raise NotImplementedError(
                f"message type not supported"
            )
    return cycles*clk_period
```

Figure 5: Interfaces for default ProtoAcc (top) and an alternative configuration of ProtoAcc (bottom). We speculate that the frequency of the alternative ProtoAcc configuration could be 2GHz (instead of 1.8GHz at the top) because the design is simpler.

```
freq = 250*10**6 #250MHz
clk_period = 1/freq
num_pe = 4
@perf_interface
def latency_darwin_gact(dna_pairs):
    cycles = (dna_pairs.ref_dna_length + num_pe + 2)*
     dna_pairs.query_dna_length/num_pe
            + num_pe + 2 + 3*dna_pairs.steps
    return cycles*clk_period
```

Figure 6: Latency interface for Darwin GACT for DNA alignment.

```
freq = 1*10**9 # 1GHz
clk_period = 1/freq
@perf_interface
def latency_menshen(pkts):
    if pkts.type == 0:
        # length of the packets stream is 100
        cycles = max(1320, pkts.sum_nr_words + 176)
    else:
        ...
    return cycles*clk_period
```

Figure 7: Latency interface for Menshen.

Menshen. The interface for Menshen is extracted per packet-stream with a fixed number of packets but of different sizes. We do not show interfaces for VTA because (unlike the other accelerators) it is a programmable domain-specific processor, so it takes "programs" as input. VTA instruction sequences contain thousands of instructions produced by compiling a high-level program with TVM [15]. These performance interfaces are therefore program-dependent and long. We expect developers to use other ltc tools instead of reading these.

5.3.2 Performance interfaces are accurate

We report in Table 3 the accuracy of the ltc-generated performance interfaces for latency. As a baseline, we use the Verilator cycle-accurate simulator to run the workloads on the accelerators' RTL. We compare the prediction provided by the performance interfaces to the values reported by Verilator. The performance interface for Menshen is only evaluated using the 100-packet testbench; the other testbenches contain too few packets to fill the pipeline, so lpn2pi's assumptions don't hold. Of course, this does not affect the LPN's accuracy (§5.2).

Accelerator	Prediction error	
	Average	Max
JPEG	7.04%	23.39%
Protoacc	2.40%	3.83%
Darwin	0.05%	0.06%
Menshen	9.43%	9.43%
VTA	19.49%	58.93%

Table 3: Prediction accuracy of extracted performance interfaces.

The average relative error is low (<20%) for all five accelerators, despite performance interfaces being approximate. They aim to capture the major factors that affect latency, not predict precisely the latency, and (as discussed in §4.2) lpn2pi introduces some inaccuracies.

The extracted performance interfaces for JPEG and VTA have the largest maximum errors. As already explained, lpn2pi does not capture the influence of bottleneck shifts on latency (§4.2). In the JPEG decoder, the input is a stream of image blocks. If one segment of blocks is highly compressed and another is less compressed, the bottleneck for processing segments of blocks will shift back and forth within the accelerator. Similarly, in VTA, each of the parallel components (fetch, load, compute, or store) can be the bottleneck during different periods while processing the instructions.

In future work, we plan to extract a performance interface for each phase of the input stream and add the latencies spent in each phase to derive the final start-to-end latency.

5.3.3 LPN-based performance simulation is up to 3 orders of magnitude faster than existing simulators

Another set of questions is "How do I generate code optimized for accelerator X", "How can I do that quickly, in compile-and-run cycles typical of software development workflows", and "How can I evaluate my envisioned workload on an accelerator that isn't available just yet?" These questions might be relevant directly to developers, or to tools, such as the TVM compiler for deep-learning models mentioned in §1. A common approach to answer such questions, when the real hardware is not available, is to use cycle-accurate simulators.

lpn2sim provides substantial benefits, up to three orders of magnitude. Fig. 8 shows lpn2sim's speedup over Verilator. All cycle-accurate simulators simulate both performance and functionality, which is wasteful when only performance

questions are being asked. Speedups are more significant with larger accelerators, because there is more functionality that the underlying LPN abstracts away.

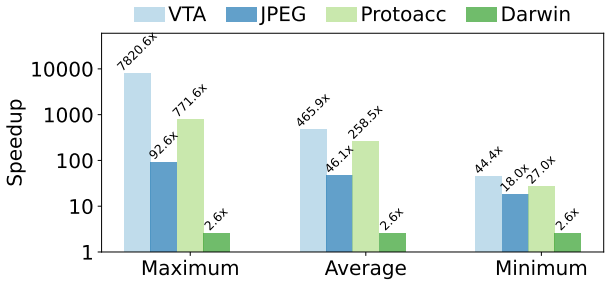

Figure 8: Simulation speedup: LPN-based simulation vs Verilator.

Table 4 shows the absolute simulation times. We believe that orders-of-magnitude changes in performance simulation time, such as going from ∼2 hours to ∼20 seconds, can bring about qualitative changes in how the tools are used.

Accelerator	Simulation time	
	Cycle-accurate Verilator	LPN-based lpn2sim
VTA	119 min	19 sec
JPEG	2159 min	38 min
Protoacc	25 sec	0.08 sec
Darwin	0.13 sec	0.05 sec

Table 4: Simulation time: LPN-based simulation vs. Verilator.

To get a feel for the impact of faster simulation time on developer productivity, we benchmark the auto-tuning process in the TVM compiler, which optimizes deep-learning models for accelerator targets (§5.1). Auto-tuning can be done either upon initial compilation, or be manually triggered whenever there are changes to the model, to the hardware, or to its configuration. We compare end-to-end compilation time when TVM uses Verilator vs. lpn2sim. Fig. 9 shows the outcome for the 10 auto-tune tasks in our workload (§5.1). As part of this auto-tuning, TVM generates 1,500 sequences of instructions, ranging from 62 to 159,947 instructions.

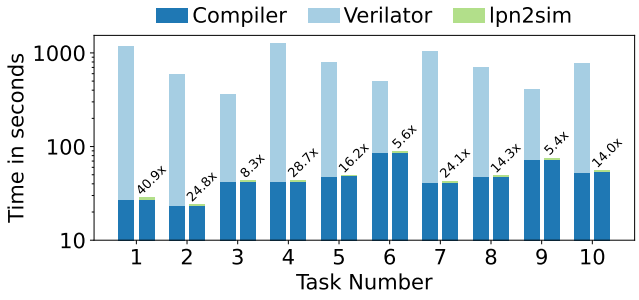

Figure 9: End-to-end compilation time, including auto-tuning. On top of the compiler+lpn2sim bars we overlay the compiler+lpn2sim vs. compiler+Verilator speedup. Despite the log-scale y-axis, one needs to zoom in to see the small amount of time taken by lpn2sim.

lpn2sim reduces auto-tuning time to a negligible amount, turning a highly *non*-interactive process into an interactive compile-and-run cycle. This enables software engineers to think differently about optimization, and to do it more often and without the accelerator hardware. Even if engineers had access to the actual hardware accelerator, using lpn2sim to auto-tune allows many more developers to do so in parallel. Compared to cycle-accurate simulation, it saves not only time but substantial amounts of compute resources and energy.

5.3.4 LPN-based tools enable performance verification

Consider the JPEG decoder in the autonomous driving scenario described in §1. To ensure safe operation in all circumstances, engineers need hard guarantees on the accelerator's performance, particularly for unseen and untested workloads. lpn2smt makes it possible to prove non-trivial bounds that are difficult to infer from source code or semantic interfaces.

The first example is determining, for some image compression ratio x%, the worst-case and best-case latency, and the corresponding worst-case and best-case inputs. Take some specific examples that may be relevant to the engineers: For a typical 90% compression ratio, lpn2smt proves that the worst-case decoding latency is 2,290 cycles for 12 RGB (18 YCrCb) 8×8 macro blocks. If the input images consist of 4 least-compressed and 14 maximally-compressed macro blocks, the best-case decoding latency is 1,717 cycles, and the difference between worst-case and best-case latency is 33%. At a 75% compression ratio, the worst-case and best-case decoding latencies are 3,063 and 2,540 cycles, respectively. lpn2smt took less than 2 minutes to find and prove these bounds.

Similarly, a Protoacc user may wonder about such bounds for serializing a message with a fixed number of bytes. We used lpn2smt to prove that, for message types with 16 fields (total 10KiB), the latency is between 726 and 1,074 cycles. SoC designers could leverage this kind of proofs when incorporating third-party accelerator blocks into their design and reason about performance implications.

lpn2smt can also be used to prove bounds on the accuracy of the performance interfaces produced by lpn2pi. Using lpn2smt, we verified formally that the latency predicted by JPEG's performance interface will always be within at most 43% of the LPN's prediction for 12 RGB (18 YCrCb) 8×8 macro blocks. This result is significant, because the input space is 64^{18} possible images, and thus infeasible to explore directly. This bound is not tight, but guaranteed to be correct.

Of course, the strength of the guarantees depends on the accuracy of the LPN. Validation tools (see §6 below) could provide confidence levels to accompany vendor-provided LPNs.

6 Discussion

In this section, we present further thoughts on how LPNs can help accelerator vendors, whether LPNs leak intellectual property, and how LPNs can be validated against the RTL.

Using LPNs in the accelerator design stage. An LPN can be written even before the accelerator's RTL is finalized. This LPN can be released to software engineers in the same organization, who can then start optimizing software for the accelerator using lpn2sim, as well as identify mismatches in performance expectations early (using lpn2pi and lpn2smt), before the design is finalized. Since accelerator vendors often release SDKs along with their accelerators, the LPN can help speed up development by providing visibility into the expected performance behavior of the accelerator before it is built.

How much proprietary information does an LPN reveal? To ensure that LPNs can be shared beyond the same organization and with software developers at large, they must not leak proprietary information. We argue that this is the case, since (1) most of the information revealed through the structure of the LPN is typically already revealed in architectural block diagrams that are made public by vendors, and (2) while LPNs provide additional information about the latency of the different compute stages, they do not describe how the accelerator achieves this latency, nor give circuit-level details and micro-architectural implementation details that are central to achieving competitive frequency and power consumption. That said, concerned vendors could still provide lower time-resolution LPNs, i.e., LPNs with coarser-grained delay functions; this reduces accuracy to safeguard proprietary details.

Validating LPNs. Since LPNs are distilled manually, they can contain mistakes; hence, after being constructed, LPNs should be validated. Developers could validate the LPN against the RTL using their RTL testbenches. Validating an LPN against the RTL is similar to how engineers validate the RTL itself using functional simulators, code reviews, and testbenches. Nevertheless, we plan to pursue building automated tools that can *formally* validate LPNs against the RTL.

7 Related Work

Petri nets have long been used to model and evaluate the performance of systems [24]. Furthermore, languages modeling a system of queues and actors are not a new idea. Kahn networks [52], dataflow networks [3, 23], and synchronous languages [8] share similarities with LPNs: more or less explicitly describing the flow of tokens in the system.

Analytical modeling of accelerators: Amid the rise of domain-specific accelerators and the need for efficient code generation, research explored semi-analytical modeling for performance models of Domain Specific Accelerators (DSA). For example, to search for good tiling and mapping of loop nests on dense tensor accelerators, [65] proposed performance models that can quickly evaluate the performance of running various loopnests on a family of accelerators. [62] tackles a similar problem for sparse tensor accelerators, and [33] for a SmartNIC. Those approaches use domain-specific knowledge in their modeling, so they typically don't offer abstractions or methodologies that can be reused in other domains. LPNs are domain-agnostic and provide a general substrate for building performance models of accelerators. There are also analytical models for accelerators that focus on data movement costs or asynchronous operations with the CPU [1, 19, 75], rather than the performance of the accelerator itself. Those models have a coarser modeling granularity than LPNs.

Performance models in the hardware community: The monograph [26] covers performance modeling techniques in detail. Analytical models based, for example, on Amdahl's law have studied various computing scenarios to establish performance trends [27, 36]. Similarly, the roofline model [84] allows simple modeling to compute performance upper bounds. Other analytical models [56, 57] build good predictors of processor performance from a few numbers: number of cache misses, branch mispredictions, etc. Finally, interval simulation [14, 30, 37] measures the distribution of performance-structuring events (cache misses and mispredictions) and profile the performance of the machine around those events to produce performance models. The way we construct performance interfaces from LPNs leverages similar principles.

Machine learning has been used to produce so-called predictive performance models of systems [25, 41, 47, 72]. These models are incomplete representations of performance, as they can only answer the questions they were trained on.

The use of simulators [10, 39, 76] to model performance is a battle-tested strategy. To make simulation faster, sampled simulation has been proposed [83, 85]. Challenges include computing warm states (caches, predictors, etc.) and identifying representative parts of benchmarks [6, 38, 67]. Finally, FPGAs [18, 54, 77, 78] can be used to speed up simulation, but FPGA simulation is possible only when the RTL is available, and compilation for FPGA is slow.

In contrast to these approaches, LPNs not only produce accurate *executable* models of hardware but can also be transformed into other useful representations (such as performance interfaces) to address broader performance questions.

LPNs for accelerators are complementary to host simulators like gem5 [10]. One can replace the gem5+RTL simulation mode with gem5+LPN for accelerators. gem5+RTL is normally bottlenecked by the RTL simulation, and gem5+LPN would shift the bottleneck to gem5.

8 Conclusion

Performance interfaces promise to offer a standardized view of accelerator performance. Despite the complexity of accelerators and system software, the LPN IR we propose can accurately represent the dynamics of various accelerators, and ltc can answer non-trivial and valuable performance questions.

9 Acknowledgments

We are grateful to Katerina Argyraki, Ed Bugnion, Jim Larus, and Diyu Zhou for their help in shaping the ideas presented here. We thank our shepherd, Abhishek Bhattacharjee, and the anonymous reviewers for their help in improving our paper.

References

[1] Altaf, M. S. B., and Wood, D. A. LogCA: A high-level performance model for hardware accelerators. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 375–388.

[2] Apache Versatile Tensor Architecture. https://tvm.apache.org/vta.

[3] Arvind, Gostelow, K. P., and Plouffe, W. Indeterminancy, monitors, and dataflow. In *Symp. on Operating Systems Principles* (1977).

[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] Baddouh, C. A., Khairy, M., Green, R. N., Payer, M., and Rogers, T. G. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2021).

[7] Beamer, S. A case for accelerating software rtl simulation. In *IEEE Micro Journal* (2020).

[8] Berry, G., and Gonthier, G. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.* (1992).

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

[10] Binkert, N. L., Beckmann, B. M., Black, G., Reinhardt, S. K., Saidi, A. G., Basu, A., Hestness, J., Hower, D., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Altaf, M. S. B., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).

[11] Boonstoppel, P., Cadar, C., and Engler, D. R. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[12] Bosshart, P., Gibb, G., Kim, H., Varghese, G., McKeown, N., Izzard, M., Mujica, F. A., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Conf.* (2013).

[13] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[14] Carlson, T. E., Heirman, W., and Eeckhout, L. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis* (2011).

[15] 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).

[16] 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).

[17] 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).

[18] Chiou, D., Sunwoo, D., Kim, J., Patil, N. A., Reinhart, W. H., Johnson, D. E., Keefe, J., and Angepat, H. FPGA-Accelerated Simulation Technologies (FAST): Fast, full-system, cycle-accurate simulators. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2007).

[19] Culler, D. E., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation. In *Proc. of Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993).

[20] Darwin: A co-processor for long read alignment. https://github.com/yatisht/darwin. Accessed 1-Dec-2023.

[21] Darwin. Darwin test data. https://github.com/yatisht/darwin/tree/master/RTL/GACT/test_data. Accessed 1-Dec-2023.

[22] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[23] Dennis, J. B. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974* (1974), B. J. Robinet, Ed., Lecture Notes in Computer Science.

[24] Diallo, O., Rodrigues, J. J., and Sene, M. Chapter 11 - Performance evaluation and Petri nets. In *Modeling and Simulation of Computer Networks and Systems*, M. S. Obaidat, P. Nicopolitidis, and F. Zarai, Eds. Morgan Kaufmann, 2015.

[25] Dubach, C., Jones, T. M., and O'Boyle, M. F. P. Microarchitectural design space exploration using an architecture-centric approach. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2007).

[26] Eeckhout, L. *Computer Architecture Performance Evaluation Methods*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[27] Esmaeilzadeh, H., Blem, E. R., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Intl. Symp. on Computer Architecture* (2011).

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

[29] 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).

[30] Genbrugge, D., Eyerman, S., and Eeckhout, L. Interval simulation: Raising the level of abstraction in architectural simulation. In *Intl. Symp. on High-Performance Computer Architecture* (2010).

[31] Google HyperProtoBench. https://github.com/google/HyperProtoBench. Accessed 1-Dec-2023.

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

[33] Guo, Z., Lin, J., Bai, Y., Kim, D., Swift, M., Akella, A., , and Liu, M. Lognic: A high-level performance model for SmartNICs. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2023).

[34] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition* (2016).

[35] Hill, M., and Janapa Reddi, V. Gables: A roofline model for mobile socs. In *Intl. Symp. on High-Performance Computer Architecture* (2019).

[36] Hill, M. D., and Marty, M. R. Amdahl's law in the multicore era. *Computer* (2008).

[37] Huang, J.-C., Lee, J. H., Kim, H., and Lee, H.-H. S. GPUMech: GPU performance modeling technique based on interval analysis. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2014).

[38] Huang, J.-C., Nai, L., Kim, H., and Lee, H.-H. S. TBPoint: Reducing simulation time for large-scale gpgpu kernels. In *Intl. Parallel and Distributed Processing Symp.* (2014).

[39] Hughes, C. J., Pai, V. S., Ranganathan, P., and Adve, S. V. RSIM: Simulating shared-memory multiprocessors with ILP processors. *Computer* (2002).

[40] Intel QAT: Accelerating data compression and encryption. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[41] Ipek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M. Efficiently exploring architectural design spaces via predictive modeling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

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

[43] Iyer, R., Argyraki, K., and Candea, G. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *Symp. on Operating Sys. Design and Implem.* (2024).

[44] 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).

[45] Iyer, R. R., Ma, J., Argyraki, K. J., Candea, G., and Ratnasamy, S. The case for performance interfaces for hardware accelerators. In *Workshop on Hot Topics in Operating Systems* (2023).

[46] Jensen, K. Coloured Petri nets: Basic concepts, analysis methods and practical use. *EATCS Monographs on Theoretical Computer Science* (1995).

[47] Joseph, P. J., Vaswani, K., and Thazhuthaveetil, M. J. A predictive performance model for superscalar processors. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2006).

[48] 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).

[49] 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).

[50] Kaggle. Div2k jpeg image dataset. https://www.kaggle.com/datasets/mingyuouyang/div2k-jpeg-0400.

[51] Kaggle. Flickr image dataset. https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset.

[52] Kahn, G. The semantics of a simple language for parallel programming. In *Information Processing, Proceedings of the 6th IFIP Congress 1974* (1974).

[53] 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).

[54] Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R. H., Bachrach, J., and Asanovic, K. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Intl. Symp. on Computer Architecture* (2018).

[55] Kim, M. A., and Edwards, S. A. Computation vs. memory systems: Pinning down accelerator bottlenecks. In *Intl. Symp. on Computer Architecture* (2010).

[56] Lee, B. C., and Brooks, D. M. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

[57] Lee, B. C., Collins, J. D., Wang, H., and Brooks, D. M. CPR: Composable performance regression for scalable multiprocessor models. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2008).

[58] Li, M., Zhang, M., Wang, C., and Li, M. AdaTune: Adaptive tensor program compilation made efficient. In *Advances in Neural Information Processing Systems* (2020).

[59] Liu, J., Maltzahn, C., Ulmer, C. D., and Curry, M. L. Performance characteristics of the BlueField-2 SmartNIC. https://arxiv.org/abs/2105.06619, 2021.

[60] 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).

[61] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. SymPy: symbolic computing in Python. *PeerJ Computer Science 3* (Jan. 2017), e103.

[62] Nayak, N., Odemuyiwa, T. O., Ugare, S., Fletcher, C. W., Pellauer, M., and Emer, J. S. TeAAL: A declarative framework for modeling sparse tensor accelerators, 2023.

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

[64] 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).

[65] Parashar, A., Raina, P., Shao, Y. S., Chen, Y.-H., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., and Emer, J. S. Timeloop: A systematic approach to DNN accelerator evaluation. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software* (2019).

[66] PCI Express Technology. https://www.mindshare.com/files/ebooks/PCI%20Express%20Technology%203.0.pdf.

[67] Perelman, E., Hamerly, G., Biesbrouck, M. V., Sherwood, T., and Calder, B. Using SimPoint for accurate and efficient simulation. In *ACM SIGMETRICS Conf.* (2003).

[68] Performance interfaces (project website). https://dslab.epfl.ch/research/perf.

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

[70] 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).

[71] Protocol buffers. http://code.google.com/p/protobuf/. Accessed on 1-Dec-2023.

[72] Qiu, Y., Xing, J., Hsu, K., Kang, Q., Liu, M., Narayana, S., and Chen, A. Automated SmartNIC offloading insights for network functions. In *Symp. on Operating Systems Principles* (2021).

[73] 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).

[74] Smith, J. E. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst. 2*, 4 (1984), 289–308.

[75] Sriraman, A., and Dhanotia, A. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 733–750.

[76] Sánchez, D., and Kozyrakis, C. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Intl. Symp. on Computer Architecture* (2013).

[77] Tan, Z., Qian, Z., Chen, X., Asanovic, K., and Patterson, D. A. DIABLO: A warehouse-scale computer network simulator using FPGAs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2015).

[78] Tan, Z., Waterman, A., Avizienis, R., Lee, Y., Cook, H., Patterson, D. A., and Asanovic, K. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Design Automation Conf.* (2010).

[79] 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).

[80] Ultra-Embedded. High-throughput JPEG decoder. https://github.com/ultraembedded/core_jpeg. Accessed 1-Dec-2023.

[81] Veripool. The Verilator simulator. https://www.veripool.org/verilator/. Accessed 1-Dec-2023.

[82] Wang, T., Yang, X., Antichi, G., Sivaraman, A., and Panda, A. Isolation mechanisms for high-speed packet-processing pipelines. In *Symp. on Networked Systems Design and Implem.* (2022).

[83] Wenisch, T. F., Wunderlich, R. E., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. C. SimFlex: Statistical sampling of computer system simulation. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2006).

[84] Williams, S., Waterman, A., and Patterson, D. A. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (2009).

[85] Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Intl. Symp. on Computer Architecture* (2003).

[86] Zuberek, W. Timed Petri nets definitions, properties, and applications. *Microelectronics Reliability* (1991).

A Artifact Appendix

Our work is part of the umbrella project on performance interfaces [68], which aims to increase performance clarity for systems code and hardware, as well as formally prove performance properties. All artifact materials are available on the project website.

The artifact has three parts: LPN, ltc, and the benchmarks. Both LPN and ltc are available through an `lpnlang` Python package. For the first part, LPN includes several constructs for developers to build their own LPNs. The LPNs we built also serve as examples of how to use those constructs. For the second part, we provide simple scripts to run ltc, which includes using lpn2pi to extract executable performance interfaces, using lpn2smt to verify performance properties, and using lpn2sim for fast performance simulation. Note, a new LPN may contain structures that cannot be analyzed by the tools in ltc; in such cases, ltc will raise an error. Automatically transforming a general LPN as defined in §3 to an LPN amenable to analysis by ltc's tools is left as a future work.

In the third part, we provide benchmarks that are used in the paper and scripts to easily run them. Note that, to simulate the Protoacc RTL, the whole SoC needs to be simulated, and the simulation is slow. The simulation speed we provided in §5 for the Protoacc RTL does not include the SoC simulation. The artifact contains also a Docker image with a ready environment to run the tools and the benchmarks. As LPN simulation normally runs for a very short amount of time, reproducing the results requires that the hardware be kept stable: disable hyperthreading, disable CPU frequency variation, etc.

We will be updating LPNs to improve readability, usability, simulation speed, and accuracy. ltc is also subject to updates that improve accuracy of lpn2pi and/or the time it takes lpn2smt to prove/disprove performance properties. The ltc toolchain will evolve to include more tools and improvements.

A.1 Formal definition of an *input class*

To fully understand how input classes are implemented and used in ltc, we present here a more detailed definition to complement that of §4.1.

A trace e of an LPN is a sequence of transition-commit records, represented as tuples $\langle T_{id}, K_I, K_O, s\rangle$. A trace characterizes the outcome of executing an LPN (as per the definition of "execution" in §4.1). The first component of a record is the transition T_{id} whose commit was recorded. The second and third component is the input set K_I, respectively output set K_O, of token IDs corresponding to the tokens consumed (respectively produced) by the commit of T_{id}. Once a token is consumed, it vanishes forever, so a token identifier can appear at most twice in a trace: as part of the commit that produced it and (possibly) as part of the commit that consumed it. The fourth component of a trace record is a sequence number s that represents the transition's commit timestamp augmented with sequencing information s.t. $\langle T, *, K_O, s\rangle \wedge \langle T', K'_I, *, s'\rangle \wedge s < s' \Rightarrow$ transition T committed before transition T' (and thus K_O was available at the same time as K'_I), even if T and T' committed at the same timestamp.

In a valid LPN trace, an input token can never be consumed before it is produced, i.e., for all records $\langle T_{id}, K_I, K_O, s\rangle$ and $\langle T'_{id}, K'_I, K'_O, s'\rangle$, $s < s' \Rightarrow K_I \cap K'_O = \emptyset$

For the rest of this section, timestamps are no longer relevant, so we drop them from our notation. We define the operator $[|.|]$ that, for a given LPN, takes a set of initial input tokens and produces a trace e of the execution of that LPN. We define the relation $\sim_1$ between pairs of traces that determines if the two traces are equivalent modulo "harmless" permutations of records as follows ($l_1 +\!\!+ l_2$ concatenates sequences l_1 and l_2):

$$\text{pre} +\!\!+ [\langle T_{id_1}, K_{I_1}, K_{O_1}\rangle; \langle T_{id_2}, K_{I_2}, K_{O_2}\rangle] +\!\!+ \text{pos}$$
$$\sim_1$$
$$\text{pre} +\!\!+ [\langle T_{id_2}, K_{I_2}, K_{O_2}\rangle; \langle T_{id_1}, K_{I_1}, K_{O_1}\rangle] +\!\!+ \text{pos}$$

A permutation as shown above is harmless if (1) $id_1 \neq id_2$; (2) T_{id_2} did not consume a token produced by T_{id_1} in the corresponding commit, i.e., $K_{O_1} \cap K_{I_2} = \emptyset$ (trace validity already implies that $K_{O_2} \cap K_{I_1} = \emptyset$); and (3) T_{id_1} and T_{id_2} do not conflict, i.e., they do not share an output or an input place.

We define relation $e1 \sim e2$ as the reflexive, transitive closure of $\sim_1$ over the set of traces of a given LPN. We can now define the set of all traces that are harmless permutations of an initial trace e as $\bar{e} = \{e' | e \sim e'\}$.

Given a trace e, we abstract it by dropping all the token IDs and keeping only the cardinality of the input and output sets in each record. Formally, we obtain the abstract trace $\alpha(e) = \text{map}(\gamma, e)$ by applying the operator $\gamma(\langle T_{id}, K_I, K_O\rangle) = \langle T_{id}, |K_I|, |K_O|\rangle$ to each record in e.

We say that the inputs (i.e., sets of initial tokens) i and i' are in the same *input class* if and only if $\{\alpha(e) | e \in \overline{[|i|]}\} = \{\alpha(e) | e \in \overline{[|i'|]}\}$.

A.2 Input class separation algorithm

Both lpn2pi and lpn2smt rely on a pre-processing step that partitions a user-defined input space into input classes. This pre-processing tool employs symbolic execution [13].

Before diving into the details of the tool, we first define the concept of a *conflict-free* transition. A transition T is conflict-free if and only if its input places are not input places for any other transition, and its output places are not output places for any other transition.

The tool symbolically executes the LPN with symbolic inputs, i.e., input space defined by the user. The tool has three main steps: (1) it first groups LPN transitions into sorted strongly-connected components (SCCs). Transitions are grouped into SCCs according to the edge directions regardless of the edge functions. (2) It then iteratively commits conflict-free transitions. The timestamp of a commit is computed locally based on the tokens (locked in the input places) and on the transition's delay. After a transition becomes enabled, the tool commits it immediately without waiting for a

delay and without synchronizing with other transitions' commits. This implies that a commit with a timestamp ts_1 can be materialized before a commit from another transition with ts_2, where $ts_1 > ts_2$, The produced tokens will carry the timestamp of the commit that produced them. (3) Once there are no more conflict-free transitions to commit, it commits one conflicting transition at a time, synchronously (with priority given to transitions in SCCs preceding other SCCs); this implies that the earliest commit is materialized first.

After symbolically executing the LPN exhaustively, the tool will find multiple paths. The input constraints associated with each path defines an input class.

A.3 Estimating $\overline{g}_T$ and ε_T in lpn2pi

To complement the description in §4.2, we provide further details on how $\overline{g}_T$ and ε_T are estimated, as well as assumptions made by lpn2pi.

We first define loops and properties of loops that lpn2pi handles. A loop in LPN is an alternating sequence of places and transitions $P_n \to T_1 \to P_1 \to \ldots \to T_n \to P_n$. Assume, for simplicity, that weights of edges are constants. lpn2pi only handles loops with the following properties:

1. The loop does not fully contain another loop, i.e., no strict subset of places and transitions in this loop forms another loop.
2. The loop has one and only one place with initial tokens.
3. The loop guarantees token conservation. Without loss of generality, assume the place with initial tokens is P_n, and it has M initial tokens. Token conservation means that initial tokens in P_n flow through the transitions, potentially changing in quantity, but eventually all M tokens flow back to P_n. This completes an iteration through the loop. More formally, a loop is token conserving if $\frac{w_{T_nP_n}}{w_{P_nT_1}} \times \frac{w_{T_1P_1}}{w_{P_1T_2}} \times \ldots \times \frac{w_{T_{n-1}P_{n-1}}}{w_{P_{n-1}T_n}} = 1$, where $w_{T_iP_j}$ is the weight of the edge from T_i to P_j.

Given these assumptions, the loop delay Δ and parallel factors F_{T_i} for each transition T_i are calculated as shown below. Recall that, initially, $\overline{g}_T$ of each transition is set to $T.\delta$, and that $\varepsilon_T = N \times \overline{g}_T$, where N is the number of commits for transition T.

$$\Delta = \sum_{i=1}^{n} \left(T_i.\delta + \overline{g}_{T_i} \times \left(\frac{F_{T_i}}{C} - 1 \right) \right)$$

$$\text{with} \quad F_{T_1} = \frac{M}{w_{P_nT_1}}, \quad F_{T_2} = F_{T_1} \cdot \frac{w_{T_1P_1}}{w_{P_1T_2}}, \quad F_{T_3} = \ldots$$

$$C = \min(F_{T_1}, F_{T_2}, \ldots, F_{T_n})$$

$$\text{and} \quad \overline{g}_{T_k} = \max\left(\overline{g}_{T_k}, \frac{\Delta}{F_{T_k}} \right)$$