Towards Agile Hardware Designs with Chisel: a Network Use-case

Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, Frédéric Pétrot

▶ **To cite this version:**

Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, Frédéric Pétrot. Towards Agile Hardware Designs with Chisel: a Network Use-case. IEEE Design & Test, IEEE, In press, 10.1109/MDAT.2021.3063339 . hal-03157426

HAL Id: hal-03157426
https://hal.archives-ouvertes.fr/hal-03157426

Submitted on 3 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards Agile Hardware Designs with Chisel: a Network Use-case

Jean Bruant
OVHcloud - TIMA

Pierre-Henri Horrein
OVHcloud

Olivier Muller
TIMA

Tristan Groléat
OVHcloud

Frédéric Pétrot
TIMA

***Abstract*—Facing Distributed Denial-of-Service (DDoS) attacks that are growing in both number and intensity, cloud service providers' network stability is at stake. Mitigation systems must provide highly responsive lines of defense while handling terabits per second. Combining reconfigurability with guaranteed throughput and latency, FPGAs are recognized targets for high-speed network applications. Although traditional hardware development flows struggle to be responsive, hardware construction languages (HCLs) bring new opportunities to hardware development. This article showcases how Chisel HCL unleashes the power of agile development methodologies through three successive development iterations of a hash-table, a core network processing module in OVHcloud mitigation systems.**

■ **THE INTERNET TRAFFIC** carries indifferently legitimate and malicious packets, with among other risks, distributed denial of service (DDoS) attacks. These attacks aim at taking down a targeted cloud service –or even at unsettling a whole network infrastructure– by saturating network devices or computing resources [1]. Only best-in-class network devices are able to handle the latest record-breaking DDoS attacks that reached peak traffic rate over 1.5 terabits per second (Tbps). To stay in business, cloud service providers are compelled to protect their customers and their infrastructures. This leads to a cat and mouse game, where attackers con-

tinuously sharpen their strategies, and mitigations strategies must be continuously updated –often within days– to remain efficient.

As reconfigurable devices implementing digital circuits, FPGAs exhibit an interesting trade-off between performances and lifetime management, which makes them well-suited to build network processing systems. They indeed benefit from the digital circuits ability to accurately control both latency and throughput of implemented algorithms, while retaining update capabilities. However development is hindered by the time-consuming process of designing and implementing circuits for FPGAs using Hardware Description Languages (HDLs). To tackle this issue, previous research focuses on increasing the abstraction level of hardware development considering two main approaches [2][3]. The first path consists in expressing functionality instead of describing hardware. This includes High-Level Synthesis (HLS), where software languages are compiled into hardware architectures, and Domain-Specific Languages (DSL), where domain-dependent primitives are provided to describe the design. The second path takes an opposite direction by bringing high-level software principles into a hardware level abstraction, to provide advanced configuration and generation capabilities. This path led to the proposal of HCLs, such as Chisel [4] or PyMTL [5].

To protect our worldwide network with more than 20 Tbps connected to the Internet, at OVHcloud we have developed our own anti-DDoS protection system which successfully mitigates thousands of attacks a day. It consists of multiple custom layers featuring both high-performance FPGAs and CPUs. Our aim being to keep hardware development synchronized with software continuous improvement, HDLs have not proven to be efficient, raising a need for an improved agile hardware development flow. After first adapting our organization to increase agility, we now need to move towards more flexible hardware design tools and languages to match the pace of software development [6][7]. As our main focus is to achieve agility through design iterations, we want to apply the principle of least power, which comes with the following philosophy:

1) Complexity is your enemy
2) Do not fear refactoring
3) Do not over engineer

In this article we present how an higher abstraction can increase agility and allow an iterative process when implementing hardware network applications. Specifically this real industry-oriented use-case enables us to:

- discuss how existing higher-abstraction solutions can be leveraged and combined to improve the whole development process
- demonstrate how HCLs can benefit hardware development and help building higher architectural abstractions of circuits,
- show that adopting these solutions can also be done in an agile manner by integrating new developments in existing HDL projects,
- evaluate the quality of HCLs generated circuits in terms of latency, throughput and resource usage against equivalent HDL implementations.

Already industrially used through the RISC-V project, we selected Chisel, a Scala-embedded HCL for our experiments. We first introduce an example of network application, focusing on existing high-level development tools. We then focus on a central function, the hash-table, describe the classic cuckoo hashing algorithm and dig into its iterative implementation as a Chisel hardware module. We show that the generated hardware is on par with our current, thoroughly optimized, SystemVerilog implementation both in terms of performances and resource usage. Finally we present our analyze of the advantages and drawbacks in the use of Chisel.

NETWORK APPLICATIONS

In order to design a DDoS filtering application, we need to collect statistics to decide if a flow is legitimate. This involves passive operations on the network, such as counting the number of packets originating from a source IP address. It may also integrate active network operation, such as sending reply packets to TCP connection requests, to check if the source is real or simply flooding the network. Based on theses statistics, a decision can be taken,

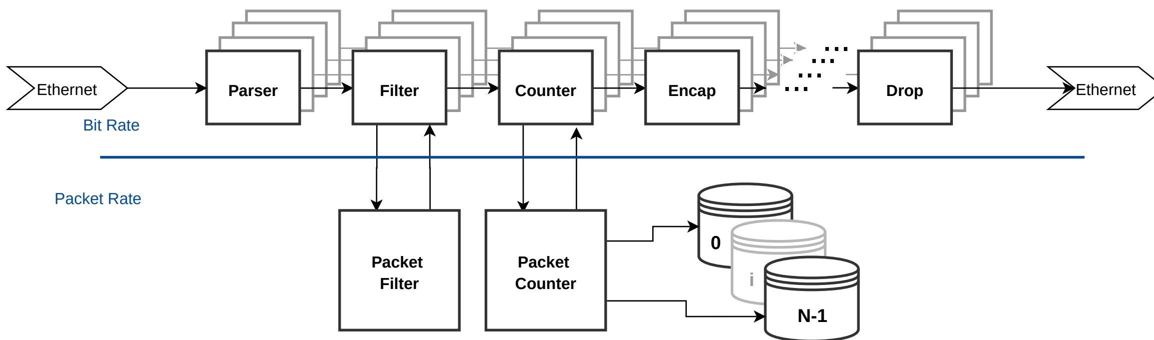

Figure 1. Network application representation

which usually consists in dropping or delaying packets from a suspicious flow. A single filtering application combines different strategies to cover the diversity of flows and attack patterns. These strategies may evolve quickly, based on new attacks or network configurations, which requires a fast development flow.

These applications also have tight performance constraints, and must guarantee worst-case performances to avoid becoming DDoS targets themselves. Throughput performances of a network application are evaluated on both bitrate and packet per second (pps) scales. The packet rate target is computed for the worst-case: smallest packets -complete Ethernet packet of 84 bytes- at full bitrate. Considering the usual bitrate target for high-end network interfaces of 100Gbps, an up-to-date network-oriented FPGA board with 8 interfaces is expected to process 1.2Gpps for 800Gbps.

In a network application, some operations such as checksum computation are done at bitrate, while others such as packet filtering or state storage are done at packet rate. This leads to a conceptually simple architecture represented in Figure 1. Network packets are split into several chunks sequentially processed in the application. Operations are chained as a pipeline for operations occurring at bit level and a sufficient number of pipelines is instantiated to reach the required bitrate. These pipelines then access shared operations executed at packet rate.

However, when going deeper into this simple architecture, complexity increases. First, the layered design of network protocols leads to complex developments and integration when new protocols need to be included. Second, network applications are stateful. For example, a rate limiter aims at limiting the incoming traffic from a given IP address. This is done by counting the flows from this address: each time a packet arrives, the IP addresses are extracted, the current number of packets is retrieved, and incremented. If the resulting number of packets is higher than a given threshold, the packet is dropped. This is a simple algorithm, but it requires to maintain the state (counter) associated with an address. The required size of state storage in the worst-case is obtained from target packet rate and retention time: 1.2Gpps with 1 second retention time requires storage for 1.2G flows.

Implementing network applications for FPGA targets with HDLs is feasible, but evolving towards faster development and deployment is desirable. DSLs are promising for defining an appliance at application level. FlowBlaze [8] is a DSL that provides a way to represent stateful applications. Another popular language is P4 [9], which is designed to program network pipelines. It relies on an architecture representation which defines the capabilities of the underlying processing system, and on an application representation based on required fields and operations applicable on packets. Toolchains implementing P4 offer real improvements over baseline HDL implementation. They automate the tedious process of protocol implementation, thus limiting the

usual errors in bit manipulation, and speeding up integration of new protocols. They also provide an implementation agnostic view of the application, which eases discussion with non-hardware specialists and helps focusing on the functionality instead of implementation details.

However, while P4 increases agility and expressiveness in this context, it is not sufficient. It relies on vendor IPs for function implementation, or on HLS when none exists. As HLS is not really adapted for these control-oriented applications, this leaves the implementation of missing functions unresolved. In our application, available state storage solutions were not satisfactory, as they were either too small, or not worst-case guaranteed. We heavily rely on QDR memories to mitigate this issue, which are not supported by the tools. The second issue with P4 is that production-ready implementations are vendor dependent. As cloud provider, we cannot depend on a single supplier and currently use both Intel and Xilinx FPGAs in our production environment. Using or developing vendor-agnostic tools could mitigate this issue, but the tight performance constraints will always lead to a need for custom implementations of critical functions. This means that agile development cannot be achieved only through the use of a network-related DSL.

CUCKOO HASH-TABLE ALGORITHM

As above-mentioned most network applications need to store per flow state. Given the range of potential source IP addresses (up to 2^{128} for IPv6), the total number of flows is far too large to use standard memory. This problem further increases when the flow is defined with additional parameters, such as traffic category, internal profile, or both source/target addresses. In practice not all slots are required, and dictionaries can be used.

While specialized Content-Addressable Memories (CAMs) or Ternary CAM (TCAMs) are perfectly suited for dictionary hardware implementation, these highly specialized circuits are ill-suited to FPGAs. Efficient dictionary implementation is usually obtained through the use of hash-tables, which can be based on external memories, decoupling logic and storage functions.

We here focus on one particular hash-table implementation, based on the cuckoo hashing algorithm [10]. This dictionary implementation provides worst-case constant lookup time while focusing on efficient memory space utilization. The present cuckoo hashing algorithm is based on N independent memory banks, respectively associated with N corresponding hashing functions. Cuckoo hashing is well suited to hardware implementation, as memory banks can be accessed in parallel.

Given a $(key, value)$ pair, the N hash functions are applied to the *key*. Each resulting hash is the address of a slot in the associated memory bank. Any $(key, value)$ pair can hence be stored in N different slots.

Given a *key*, the lookup operation is quite straightforward:

1) N hashes of the *key* are computed,
2) slots pointed by each hash are retrieved,
3) if the *key* is found in one of the retrieved slots, the associated *value* is returned.

The lookup operation execution time does not depend on where or when the data was stored, ensuring worst-case constant lookup time.

The insert operation is a bit more complex. Given a $(key, value)$ pair to be inserted, the last step is replaced by:

3) Lookup for free slots:
    a) If at least one slot is free, randomly pick one of them for insertion,
    b) Otherwise swap the content of one randomly chosen slot with the $(key, value)$ pair and go back to step 1).

The re-insert operations of overwritten $(key, value)$ pairs are called *moves*. Having *moves* create a possibly infinite loop in the design, if no free slots are found for each successive move. This is avoided by limiting the number of reinsertions, dropping the last $(key, value)$ pair when the limit is reached.

This hash-table was already implemented and used in our anti-DDoS solution, with many improvements and variations to fit our use-cases. As part of an ongoing migration of our applications from SystemVerilog to Chisel, we decided to re-implement it, starting with a very

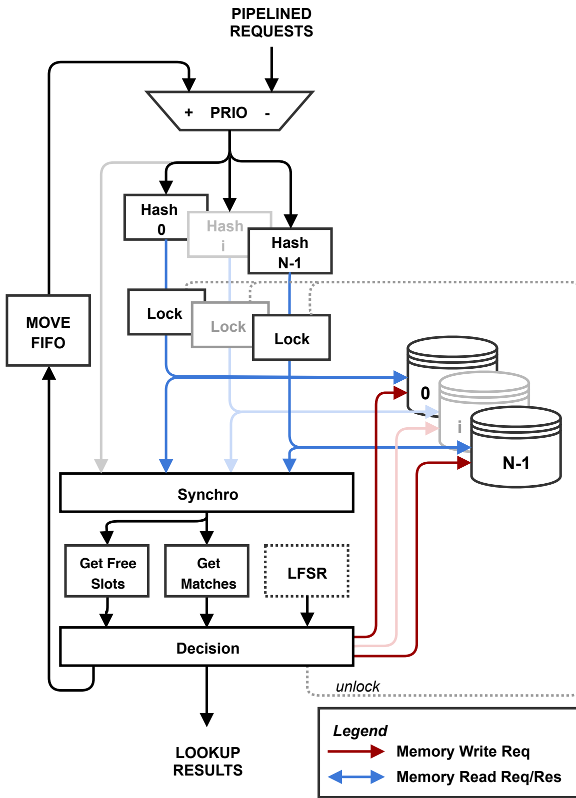

Figure 2. Global Cuckoo Hash-table Architecture

basic one, and iteratively adding the required features.

We target the following performance requirements:

- one operation per clock cycle,
- a latency within hundreds of cycles, which is quite transparent at network scale.

The first design iteration is a core cuckoo hash-table module depicted Figure 2. *Moves* are blocking, with incoming requests stalled until the move is successful or the limit is reached.

The implementation process for this module is the same with Chisel or with SystemVerilog. The hardware description closely follows the linear pipeline architecture. Expressing this simple assembly of modules from a coarse-grained point of view is straightforward in both languages. This is expected from SystemVerilog, but it validates the fact that Chisel provides the necessary constructs to be used as a standard hardware description language. It also enables the inclusion of existing SystemVerilog modules as black-boxes, a mechanism used in this initial implementation to integrate our existing custom hash functions. While there is no significant gain in using Chisel at this point, there is also no loss in hardware expressiveness.

READ-MODIFY-WRITE ITERATION

As the second design iteration, we extend the initial implementation with the ability to process user-defined *read-modify-write* operations. This allows for example to increment a counter within a single request. A *read-modify-write* operation consists of 3 successive steps:

1) lookup a $(key, value)$ pair,
2) compute updated *value* given user-defined algorithm,
3) write result back to original memory slot.

The complexity of this iteration lies in the user-defined aspect of the modification. Adding an atomic *read-modify-write* is simply a matter of locking the table and inserting the user-defined function between the decision stage and the actual write to memory. But being able to externally provide the operation is not as simple. Our SystemVerilog implementation uses an external module for this operation, and carefully designed hardware ports, but this approach comes with many pitfalls. First, increasing the number of ports increases the complexity, reducing overall expressiveness of the source code. Modifying the function means ensuring that the connection is done correctly. It also implies a complex validation of the modifier, mocking the expected in-place integration. Secondly, providing a default modifier can only be done with a wrapper module or with *generate* statements, which limits code reuse. Lastly, the resulting hierarchy seen during verification and synthesis analysis presents the modifier outside of the main hash-table pipeline, resulting in harder debug and resources enumeration. In our anti-DDoS application, we use several different hash-tables, with different modifier functions. Limitations encountered using SystemVerilog for this simple *read-modify-write* upgrade are symptomatic of design patterns that prevent code reusability with HDLs. As upgrades come at high maintainability and ver-

ification costs, full rewrite are often preferred in practice.

In contrast functional parameterization and object inheritance in Chisel provides the ability to split roles. Users can instantiate the hash-table with any modifier function, provided that it fulfills the software interfaces documented below (for simplicity, we use a *UInt* as data type).

```
trait CuckooModifier {
   def build(valid: Bool, input: UInt)
      : (UInt, Bool)
}
```

The build function is the main function called in the hash-table to generate modifier hardware. The concrete modifier implementation is then for example defined as the *Increment* modifier below:

```
object Increment extends CuckooModifier {
   def build(valid: Bool, input: UInt)
      : (UInt, Bool) = {
      (input + 1.U, valid)
   }
}
```

Finally the HashTable module takes a *CuckooModifier* as parameter, and uses it in its implementation.

```
class HashTable(
   val modifier : Option[CuckooModifier]
      = None
  // ...
) extends Module {
   //...
   modData, modValid = modifier match {
      case Some(mod) => mod.build(
         lookupValid, lookupData)
      case None =>
         (lookupValid, lookupData)
   }
   //..
}
```

Possible flavors of this parameterization are endless. For example, we could have used the *build* function as a parameter, but using a trait allows for more complex modifiers. We could also use a list of modifiers, along with a modifier selector mechanism, to implement different modifications in a single hash-table. With this approach, hardware interfaces are not growing out of control, hierarchy is preserved and a default child module generator can be provided to the module constructor, hence increasing reusability. Changes in code are quite limited and confidently integrated into the code base thanks to associated unit-tests. The flexibility offered by this parameterization hardly suffers from any limitation and greatly helps with integrating the Chisel implementation within our complex code base.

EXPERIMENTING WITH HASH FUNCTIONS

The hash function choice is a balance between resource usage and latency on one hand, and collision avoidance on the other hand. While redesigning our hash-table, we wanted to experiment with other hash functions, which is the subject of this third iteration.

In SystemVerilog, new hashes can be integrated in several ways: using hardware ports as above-mentioned; manually selecting the desired hash function within hash-table code; or with a generic parameter, coupled to an enumeration of known hash functions using *generate* statements. However, none of these solutions avoid modification of original code for each new hash function -or at the cost of complex and constrained interfaces. In Chisel, as described through the second iteration, functional parameterization allows fast integration of user-defined code without internal changes, thus easing design exploration and increasing reuse.

An interesting hash function candidate is the SipHash algorithm [11] which was deemed one of the most performant hash [12]. Originally designed for software, it is highly sequential with multiple iterations of a computation –the *sipround*– over the input message. To integrate SipHash in our fully-pipelined architecture we need to unroll the sequential loops while duplicating the *siprounds* and chaining them through register stages. The number of *siprounds* is configurable and impacts the hash function properties. Within SystemVerilog, the usual way to deal with such variable length pipelines is to define an array of register stages. These stages are then connected using their respective indexes which results in low read-

ability and highly error-prone code. Resorting to a function to factorize *sipround* computations between registers stages is appealing. Unfortunately, SystemVerilog functions can hardly integrate registers, which prevents exploration of different registers configuration inside the stages.

In contrast Chisel supports recursive generating functions which are able to describe both the functionality and the pipeline details by inferring either registers or wires between stages. Hardware stages are defined during elaboration (*i.e.* the execution of these generating functions), leading to a very comprehensive match between the algorithm steps and the generated hardware stages.

This third iteration shows two other aspects of the usefulness of Chisel: the extended use of functions for both code generation and advanced software configuration. First, the possibility to instantiate hardware inside a function allows more natural expression of algorithms. Secondly, configuration functions can be defined as software within the hardware description, whereas for our SystemVerilog hash functions, a configuration file is generated using a Python script prior to hardware elaboration. This limits the number of errors, and interestingly speeds up simulation and synthesis, as the hardware is generated in a static form, as opposed to the SystemVerilog version where configuration file is read dynamically by the tools.

HARDWARE EVALUATION

The primary goal of languages is to allow efficient implementation in terms of resource usage and performance. In resources-constrained FPGA, a language must be chosen carefully in this respect. As we observed the same trend over numerous possible parameterizations, this section focuses on the result corresponding to the following configuration:

- 75 bits key width
- 21 bits address width
- 69 bits data width
- frequency: 200 MHz
- device: Xilinx VU9P

Table 1 summarizes the resource usage for

		Base	+ SipHash	+ *increment*
Verilog	LUT	9162	22043 (+12881)	22215 (+172)
	LRAM	0	1679 (+1679)	1679 (+0)
	FF	16705	24742 (+8037)	25135 (+393)
	BRAM	11.5	11.5	11.5
Chisel	LUT	9266	22371 (+13105)	22441 (+70)
	LRAM	145	1960 (+1815)	1960 (+0)
	FF	15393	23636 (+8243)	23806 (+170)
	BRAM	11.5	11.5	11.5

Table 1. Hardware resource usage comparison through iterations

both SystemVerilog and Chisel implementations at each iteration. We took care to implement similar pipeline in terms of register stages. This shows that even if SystemVerilog allows to spare a few resources, both technologies display comparable usage. The difference comes from slightly different design choices, and from the randomization of hash functions. Overall, using Chisel does not come at a significant cost in resources, which is remarkable given the maturity of the SystemVerilog implementation.

Moreover, synthesis time, omitted in this table, is up to 5 times lower for Chisel-generated Verilog than for SystemVerilog. This is mainly due to the fact that no generation occurs within Chisel-generated Verilog, which limits the required operations -and possible tool-related errors. Ease of use is also enhanced: a single file is generated for simulation, which can then be included directly in the synthesis tools for synthesis.

AGILE-FRIENDLY DESIGN

As demonstrated with this study, Chisel parameterization process greatly improves the flexibility and reusability of modules. While agility is achievable with SystemVerilog, iterations are limited by the low genericity of the language thus impacting the entire flow. Low flexibility leads to harder iterations, which in turn lead to bigger increments, as the integration of a feature overwhelms the time required to develop the said feature. When using Chisel, the entire focus is on small increments.

This improved agility also applies to the design and validation flow, presented in Figure 3, through the elaboration step. In SystemVerilog,

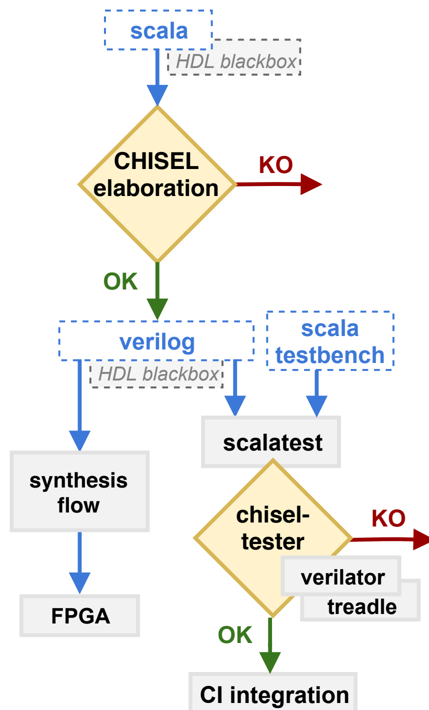

Figure 3. Chisel design and validation flow

each simulation or synthesis tool has its own elaboration process, supporting its own subset of the language. Simulation tools are usually more feature-rich, and synthesis tools limitations are found late in the flow, requiring a full rewrite of the already validated module. To avoid this, we usually stick to a very basic subset of the language, further limiting code reuse. Chisel elaboration ensures transformation of Chisel code to this basic subset. It also provides numerous checks such as types or combinational loop checking. As it occurs before functional simulation, this flow brings three major benefits. First, these checks do not need to be asserted during functional validation anymore. Secondly, many mistakes can be caught in a matter of seconds before any lengthy simulation start-up, which reduces iteration times and improves developer experience. Thirdly, the generated Verilog is compatible with all synthesis and simulation tools.

On the validation side, *ScalaTest* library natively provides test-cases management and straightforward integration of pass/failed results into continuous integration (CI) systems. It allows testing of collections of programmatically defined parameter sets. For example, it's possible to routinely check all existing hash functions within a single test. Testbenches, also written in Scala, are fed to the Chisel-tester which provides bindings with fast open-source hardware simulators such as *treadle* and *verilator*.

In conclusion HCLs introduce more checking steps, taking place earlier in the validation flow, enabling faster iterations and hence improving the overall hardware development agility and predictability.

CONCLUSION

As designers of hardware network applications, we seek to improve development through agility. While the emerging DSLs such as P4 are promising for functionality description, the need for custom hardware functions remains. We developed a hash-table using Chisel as construction language and evaluated the contributions of the agility it provides. The high-level constructs Chisel brings allow an actual iterative development, permitting small increments, with improved validation at each step. In the end, our hash-table was successfully integrated into the OVHcloud anti-DDoS mitigation system, with comparable resource usage and performance.

Designing generators instead of describing circuits proves to be an efficient approach which fully exhibits its agility as soon as successive design iterations occur. HCLs integrate powerful software engineering concepts into hardware development, unlocking higher abstraction levels while still mastering generated hardware. As a direct benefit, it becomes easier to build highly reusable design libraries. We now expect to reshuffle our vision of basic-blocks integration in hardware network applications designs.

REFERENCES

1. D. Moore *et al.*, "Inferring internet denial-of-service activity," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 115–139, 2006.
2. D. F. Bacon *et al.*, "FPGA programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

3. O. Shacham *et al.*, “Rethinking digital design: Why design must change,” *IEEE Micro*, vol. 30, no. 6, pp. 9–24, 2010.
4. J. Bachrach *et al.*, “Chisel: constructing hardware in a scala embedded language,” in *Proc. of the 49th Design Automation Conference*. IEEE, 2012, pp. 1212–1221.
5. D. Lockhart *et al.*, “Pymtl: A unified framework for vertically integrated computer architecture research,” in *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 280–292.
6. Y. Lee *et al.*, “An agile approach to building risc-v microprocessors,” *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.
7. A. Izraelevitz *et al.*, “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations,” in *Proc. of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 209–216.
8. S. Pontarelli *et al.*, “Flowblaze: Stateful packet processing in hardware,” in *Proc. of the 16th USENIX Symposium on Networked Systems Design and Implementation*, 2019, pp. 531–548.
9. P. Bosshart *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
10. R. Pagh and F. F. Rodler, “Cuckoo hashing,” *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
11. J.-P. Aumasson and D. J. Bernstein, “Siphash: a fast short-input PRF,” in *Proc. of the International Conference on Cryptology in India*. Springer, 2012, pp. 489–508.
12. W. So *et al.*, “Named data networking on a router: Fast and DoS-resistant forwarding with hash tables,” in *Proc. of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 2013, pp. 215–226.

Jean Bruant is currently with OVHcloud as PhD candidate in partnership with the TIMA Lab of Univ. Grenoble Alpes (UGA). His research interests include hardware description languages, EDA tools and network acceleration with FPGAs. He received a M.Sc. in electrical engineering in 2018 from IMT Atlantique. Contact him at jean.bruant@ovhcloud.com

Pierre-Henri Horrein is currently with OVHcloud. His research interests are mainly focused on high-level design of hardware functions and on interaction with software. He received his Ph.D. in computer science in 2012 from Université de Grenoble. Contact him at pierre-henri.horrein@ovhcloud.com

Olivier Muller is currently with Grenoble INP/TIMA as associate professor. His research interests are in the areas of reconfigurable computing, methodologies and CAD tools for FPGA and FPGA virtualization. He received his Ph.D. in electrical engineering in 2007 from UBS (Université Bretagne Sud). Contact him at olivier.muller@univ-grenoble-alpes.fr

Tristan Groléat leads a development team at OVHcloud, building software and FPGA-based solutions which aim at protecting the OVHcloud network from DDoS attacks. He received his Ph.D. in computer science in 2014 from Telecom Bretagne. Contact him at tristan.groleat@ovhcloud.com

Frédéric Pétrot is a professor at Grenoble INP/TIMA. His research focuses on the design, implementation and tools for multiprocessor systems-on-chip and ad-hoc accelerators. He got a Ph.D. in computer science (1994) from Université Pierre et Marie Curie in Paris, France. Contact him at frederic.petrot@univ-grenoble-alpes.fr