Tydi

an open specification for complex data structures over hardware streams

Peltenburg, Johannus Willem; Brobbel, Matthijs; Van Straten, Jeroen; Al-Ars, Zaid; Hofstee, Peter

DOI
10.1109/MM.2020.2996373

Publication date
2020

Document Version
Accepted author manuscript

Published in
IEEE Micro

Department: Head
Editor: Name, xxxx@email

Tydi: an open specification for complex data structures over hardware streams

Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, and Zaid Al-Ars
Delft University of Technology

H. Peter Hofstee
IBM, Delft University of Technology

***Abstract*—Streaming dataflow designs describe hardware by connecting components through streams that transport data structures. We introduce a stream-oriented specification and type system that provides a clear and intuitive way to map complex, dynamically-sized data structures onto hardware streams. This helps designers to lift the abstraction of streaming dataflow designs, reducing the design effort. The type system allows complex data structures to be as easy to use in streaming dataflow designs as in modern software languages today.**

EXCHANGING DATA between components of a computing system is a major topic in computer architecture. When components interact, a well-specified representation of the data should exist in whatever medium used for communication to allow the data to be interpreted correctly and enable reusable and extensible designs. Clear format specifications are especially useful for an open-source community, where it enables more efficient collaboration.

Agile development of hardware-oriented solutions is driven by many excellent open-source projects that increase the level of abstraction at which hardware is described. Some experts even argue that we are in "A Golden Age of Hardware Description Languages" [1] — more advanced designs can be automatically synthesized from fewer lines of code.

However, we observe a lack of standardized exchange formats and abstract views for complex data structures at the level of digital circuits. As a result, developers often manually design their custom representations of more advanced composite and aggregate data structures (e.g. strings, nested lists, etc.), that need to be exchanged between components over streams.

We propose **Tydi**; an open specification (found freely online [2]) that allows developers to map composite and dynamically-sized data structures onto hardware streams. It furthermore provides an abstract, but still hardware-oriented view of these data structures, as to not lose the opportunity to make common trade-offs in the design phase. An overview of the context of Tydi

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2020.2996373, IEEE Micro

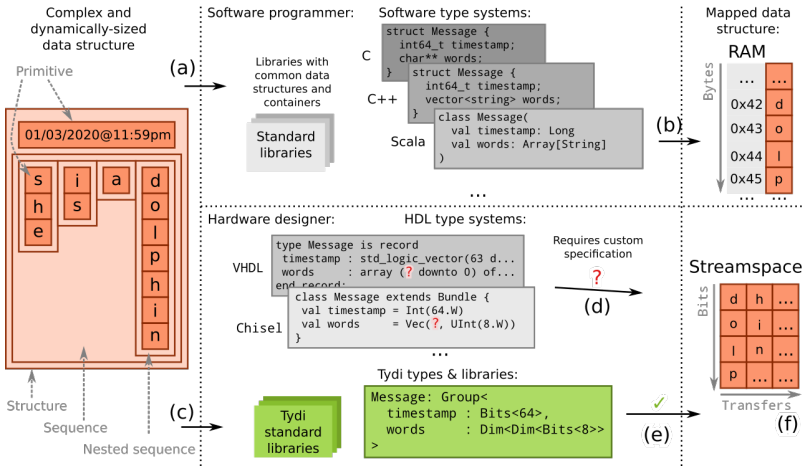


Figure 1. Tydi context. To implement a data structure, programmers choose some types and containers, helped by language constructs and libraries (a), run-time engines and compilers take care of the mapping to RAM (b). The same for contemporary HDLs (c) is prevented because dynamically-sized structures are not inherently supported. When mapping to hardware streams (d) designers customize solutions to transport data structures over multiple stream transfers. Tydi is a specification that clearly and intuitively provides a mapping (e) and pre-defined containers for common types.

is seen in Figure 1.

At the core of the specification lies a type system. It provides an intuitive and clear definition of how complex data structures are transported over hardware streams. We discuss additional parameters that can be used to make an area/throughput trade-off for component interfaces, and provide a precise specification at the hardware level. This specification can be used by developers that design components, or that combine components into larger designs, either manually or by automated tools.

BACKGROUND

Designing digital circuits from a dataflow-oriented perspective involves selecting appropriate transformations and connections between the transformations through directed channels. When data starts flowing from external sources, the specific configuration of transformations and channels allow an algorithm to be executed, producing output that can flow back to an external sink. For digital circuits, channels are often implemented as *streams*; point-to-point connections, where a sink receives data elements from a source in FIFO-order. Transformations are typically implemented as **streamlets**: components with streaming interfaces.

When data structures flow over streams between streamlets, it is favorable to reason about them at a high level of abstraction, rather than at a low level (bits and clocks), especially when the data structures are dynamic and complex. An example data structure that we will use throughout this article, is a chat message consisting of a (64-bit POSIX-time encoded) timestamp and a sentence (Extended-ASCII encoded string). To create more context, envision an application with an unbounded stream of messages, where one would like to apply a transformation that filters the message by some time range, then splits the sentence into separate words.

A more formal view of data types and structures is presented in Table 1. Using that view, we can describe the aforementioned chat message as: $T_m = \textsc{Struct}\langle\textsc{Prim}\langle 64\rangle, \textsc{Seq}\langle\textsc{Prim}\langle 8\rangle\rangle\rangle$, and the filtered message as: $T_f = \textsc{Struct}\langle\textsc{Prim}\langle 64\rangle, \textsc{Seq}\langle\textsc{Seq}\langle\textsc{Prim}\langle 8\rangle\rangle\rangle\rangle$.

In the software domain, instantiated data structures of these types are typically materialized as bytes in a RAM. How this is done depends on the software framework used, as shown in Figure 1. The exact byte-level representation is left to compilers, run-time engines and standard libraries. Especially for aggregate types, programmers typically select pre-defined *containers* from

Table 1. Conceptual view of data types and data structures used throughout this article.

Data type	Data structure (or instance)	Description
EMPTY	$(\emptyset)$	Empty set, singleton value.
PRIM$\langle B \rangle$	$(b_{B-1}, b_{B-2}, ..., b_0)$	Primitive element containing B bits of information.
STRUCT $\langle T_1, T_2, ..., T_n \rangle$	$(\mathrm{I}(T_1, p_1), \mathrm{I}(T_2, p_2), ..., \mathrm{I}(T_n, p_3))$	Composite type. An instance is a set with one instance of every type argument $T_1, T_2, ..., T_n$.
VARIANT $\langle T_1, T_2, ..., T_n \rangle$	$\mathrm{I}(t \in (T_1, T_2, ..., T_n), p_t)$	A variant type. An instance is one of either type T_1 or T_2, etc. t is known when instantiated, by some tag.
TUP$\langle n, T \rangle$	$(\mathrm{I}(T, p_1), \mathrm{I}(T, p_2), ..., \mathrm{I}(T, p_n))$	A fixed-length aggregate type. An instance is a sequence with $n \in \mathbb{N}^+$ instances of the same type T. n is part of the type.
SEQ$\langle T \rangle$	$(\mathrm{I}(T, p_1), \mathrm{I}(T, p_2), ..., \mathrm{I}(T, p_n))$	A variable-length aggregate type. An instance is a sequence with $n \in \mathbb{N}^0$ instances of the same T. n is only known when instantiated.

$\mathrm{I}(T, p)$ is an instance of type T, where p parametrizes the instance, if necessary.

standard libraries (e.g. the C++ `std::vector`) that help mapping tuples and sequences based on their basic notion of the architecture of their device (typically a load-store architecture) and properties of their algorithm/workload (e.g. whether to store a sequence as a linked-list or in a hash-table). This greatly abstracts the details of *how* the data structure is mapped onto (typically) a RAM — a one-dimensional sequence of bytes, under some constraints (the total number of bytes available), but programmers retain some control over the performance characteristics of the mapping.

While attempting to map complex and dynamically-sized data structures onto a single streamed element, one quickly finds it impractical to allow the streamed element to be as wide as the amount of information in bits. This impracticality exists for at least two reasons. First, some aggregate types, such as the sequence, are dynamically-sized. Accommodating an interface at design-time based on some initial guess for its length would rule out support for potentially larger sequences. Second, data structures described by aggregate types that *are* statically-sized, such as tuples, can grow arbitrarily large. Streamlets may not be able to absorb all data from a large element at once. Consequently, one would be under-utilizing the resources used for the streaming interface.

Thus, designers often choose to split the information over multiple stream transfers, such that over time, the whole data structure is transported between the sourcing and sinking streamlet. Therefore, a hardware developer does not map a data structure merely onto space (e.g. a one-dimensional bit-vector), but also onto time, or more specifically, stream transfers. From this description, a two-dimensional plane emerges that we will call **streamspace** — the plane consisting of both a spatial resource (bits) and a resource of temporal nature (transfers).

To the best of our knowledge, while there is an enormous body of work in the software domain about mapping complex data structures onto byte-addressed RAM, little literature exists that discusses methods of mapping composite, potentially dynamically-sized and nested aggregate types onto streamspace from an abstract point of view. This causes the tedious need for hardware designers to create custom formats for their designs and data structures (often on top of existing standards), which is a problem we address through Tydi.

RELATED WORK

One widely-used streaming protocol specification is the AXI4-Stream protocol [3]. Users can transport anywhere between zero and N bytes per transfer, with an (optional) `last` bit that denotes the end of a one-dimensional sequence of bytes. It therefore specifies how to transport either PRIM$\langle 8 \rangle$ or SEQ$\langle$PRIM$\langle 8 \rangle\rangle$. It does not specify how structures that are not byte-oriented or that have deeper levels of nesting, e.g. SEQ$\langle$SEQ$\langle$PRIM$\langle 7 \rangle\rangle\rangle$, should be communicated. Avalon Streaming [4] is similar to AXI, but slightly less restrictive, because elements can be arbitrarily sized.

CoRAM++ [5], where DMA engines are generated based on a set of specific C-style data structures, such as multidimensional arrays, linked lists, and trees, allows streamlets to interact with more advanced data structures in memory, but does not focus on communication between streamlets or on how to mix the above data structures.

We have explored active (open-source) hardware frameworks, including classical HDLs (VHDL, Verilog, SystemVerilog) and contempo-

rary ones (Cλash [6], Chisel [7], and Spatial [8]). All these HDLs support compound types that map onto bit-vectors (e.g. VHDL's `record`, Chisel's `Bundle`, etc.), and statically-sized aggregate types, but lack inherit support for dynamically-sized aggregate types mapped onto streamspace. This is unsurprising; the type systems of these frameworks reason only about space, but not about stream transfers — the latter being typically left to the designer — as the goal is to describe hardware just above the register-transfer level. In libraries of some of the languages, abstractions for streaming dataflow designs are provided, e.g. Chisel's `DecoupledIO`, Spatial's `StreamIn/Out` and Cλash's `DataFlow`. The abstractions move towards the level we envision when composing designs out of streams and streamlets, but only abstract the handshake mechanism for otherwise completely user-defined signals, lacking inherent support for throughput scaling of streams that is available in AXI/Avalon.

Commercial high-level-synthesis frameworks (including Vivado HLS and SDAccel) support streams as parameters for functions, creating a streaming interface for kernels. These streams provide an abstraction for the handshake protocol of a single unbounded stream for statically-sized composite types. Information about the size of dynamically-sized aggregate types traveling over the stream still requires a custom mapping onto the streamed elements.

ENTERING STREAMSPACE

As mentioned in the previous sections, our goal is to find a suitable mapping of the data structures shown in Table 1 into streamspace. We propose a mapping, where we define **logical streams**; streams that transport a top-level data structure (that may consist of nested data structures). Depending on the data structure, a logical stream can consist of multiple **physical streams**; streams with their own handshake/transfer interface.

To facilitate a clear definition of the physical streams emerging from a logical stream, we introduce a *streamspace*-oriented type system. The type system exposes the direction of physical streams, and how two of their properties E and D are derived. E is the number of bits of an *element* that the stream transports in every transfer, and D is the number of bits used to signal the end of some (nested) sequence. The physical streams have more properties that are explained in the next section.

At least three use-cases for this type system exist. First, it can be used in tools that automatically generate streamlet interfaces for traditional hardware description languages (e.g. VHDL or (System)Verilog). In a later section, we briefly discuss two implementations of such generators; the reference implementation utility of Tydi, and Fletcher, a hardware acceleration framework for FPGAs. Second, the type system can be used in hardware description frameworks, such as Chisel. Chisel has highly generative capabilities through its host language Scala. The type system and generative code can reside in a Scala library. Third, we envision tight integration within hardware description languages that use a functional programming paradigm, such as Cλash, as they are highly suitable to express dataflow designs.

A STREAM-ORIENTED TYPE SYSTEM

We define six types that help to construct a streamspace representation of the data structures, also shown in Table 2. These types abstract indivisible properties of data structures being exchanged in streamspace. More advanced abstractions can be constructed by combining these types, as shown in Figure 2 (discussed later).

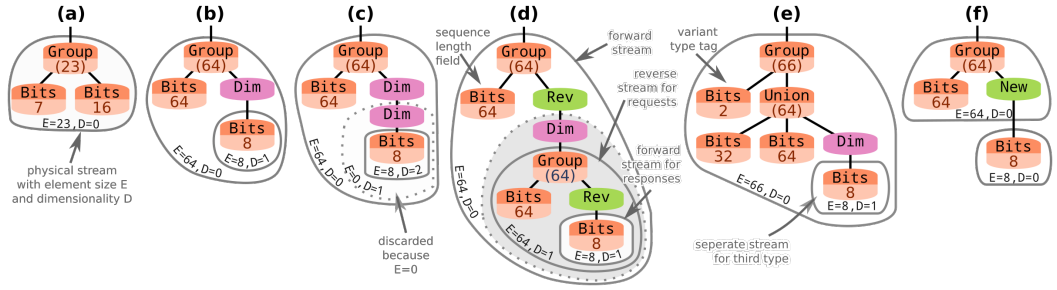
The first three types in the table manipulate the size E of the element that a physical stream transports. As such, they could 'live' outside streamspace (i.e. they map only to a one-dimensional bit vector). The other types are used to create separate physical streams in streamspace.

Of the element-manipulating types, the first, $\textsc{Bits}\langle B\rangle$ will add B bits to that element, and could be seen as simply adding a field of a primitive type to the streamed element. This is the streamspace representation of a $\textsc{Prim}\langle B\rangle$. The second, $\textsc{Group}\langle S_1, S_2, ..., S_n\rangle$, concatenates elements of its child types (where S denotes a streamspace type parameter). This causes the element size E to be the sum of all child element sizes, as long as these children reside in the same physical stream. GROUP therefore allows to represent STRUCT, but can also help to combine multiple physical streams, as the type ar-

Table 2. Overview of streamspace types in Tydi

Type	Description	D_{child}
BITS$\langle B \rangle$	Defines a B-bits primitive element, where $B \in \mathbb{N}_0$.	n/a
GROUP$\langle S_1, S_2, ..., S_n \rangle$	Concatenates elements of types $S_1, S_2, ..., S_n$ into one physical stream element.	D_p
UNION$\langle S_1, S_2, ..., S_n \rangle$	Defines a B-bits element, where B is the max. element width of $S_1, ..., S_N$	D_p
DIM$\langle S \rangle$	Creates a streamspace of elements of type S in the next **dimension** w.r.t. its parent.	$D_p + 1$
REV$\langle S \rangle$	Creates a new physical stream of S that flows in **reverse** direction w.r.t. its parent.	D_p
NEW$\langle S \rangle$	Creates a **new** physical stream in the parent space D_p with elements of type S.	D_p

D_0 is the first streamspace dimension, D_p is the dimension of the parent type, if applicable.



(a) A streamspace mapping of a structure with a seven-bit field and a sixteen-bit field: STRUCT$\langle$PRIM$\langle 7 \rangle$, PRIM$\langle 16 \rangle\rangle$. In the mapping GROUP$\langle$BITS$\langle 7 \rangle$, BITS$\langle 16 \rangle\rangle$, GROUP concatenates the BITS elements together into a single element, resulting in a single physical stream transporting twenty-three-bit elements ($E = 23$) with dimensionality $D = 0$.

(b) The type T_m of our chat message example. A simple mapping of T_m into streamspace is: GROUP$\langle$BITS$\langle 64 \rangle$, DIM$\langle$BITS$\langle 8 \rangle\rangle\rangle$ creating two physical streams; one for the timestamp field, and another, logically nested in the first, for the sequence of 8-bit elements. For every transfer on the first stream, there must be at least one (possibly empty) transfer on the second stream.

(c) Output T_f of the streamlet transforming T_m. The second field is now a *sequence of sequences*, requiring a nested DIM. Although the outer DIM defines a new physical stream, it is discarded because its element size is zero. The stream transporting the nested sequence has $D = 2$ dimensionality bits to encode the three possibilities for every element transported: it is the last element of the inner sequence but not the outer, or it is the last element of *both* sequences, or it is the last element of *neither* sequence.

(d) A type allowing *random* access to an element from a sequence SEQ$\langle$BITS$\langle 8 \rangle\rangle$. We map this to streamspace as: GROUP$\langle$BITS$\langle L \rangle$, DIM$\langle$REV$\langle$GROUP$\langle$REV$\langle$BITS$\langle L \rangle\rangle$, BITS$\langle 8 \rangle\rangle\rangle\rangle\rangle$ where L is the number of bits used to represent sequence lengths. The streamlet sourcing the random element *first* provides the length of the sequence on the outermost physical stream, so that the sink knows how large the sequence is (to prevent requesting out of bounds). Then, for every sequence length, the sink may send multiple (hence DIM) requests through a reversed (hence REV) physical stream. For every request, an element is provided (hence the GROUP of the BITS and REV). This describes a streamed RAM interface. The arguments of GROUP are strictly ordered. To prevent deadlocks, a source *may not assume* that the sink accepts transfers on streams out of the order of appearance as type arguments.

(e) An example of a mapping of the type VARIANT$\langle$PRIM$\langle 32 \rangle$, PRIM$\langle 64 \rangle$, SEQ$\langle$PRIM$\langle 8 \rangle\rangle\rangle$. The first field of the group contains the variant type *tag* to let the sink know what type of instance is contained in the variant. Because the first two potential types are bit fields, they can fit into the outermost stream through the UNION type, causing the element size to be the maximum of the size of the BITS fields, in this case $E = 64$. Since the third type has a higher dimensionality ($D = 1$), its instances flow over their own physical stream. Whenever the tag exposes that the element is of the third type, the sink must read the rest of the instance from the innermost stream.

(f) A use for NEW. Instead of mapping the length of a sequence by increasing D, we may choose to map the sequence length as a separate stream. This can be seen as another way of mapping an instance of a SEQ into streamspace.

Figure 2. Examples of streamspace types.

guments are not limited to element-manipulating types. The final element-manipulating type is UNION$\langle S_1, S_2, ..., S_n \rangle$, that selects the element size to be the largest element size of its children. This is useful in representing the VARIANT type.

Of the physical stream creating types, DIM$\langle S \rangle$ increases the *dimensionality* of its child type S, and therefore increases the parameter D. In physical streams, D bits are reserved that signal an element is the *last* element in a (nested) sequence (rather than e.g. the single 'last' bit of AXI4-Stream). A separate physical stream is created over which zero or more instances travel for every *single* element of its parent. This makes DIM$\langle S \rangle$ suitable to represent (nested) sequences. REV$\langle S \rangle$ is used to create a physical stream that flows in the reverse direction respective to its parent. This stream remains in the same dimension as its parent; for every element that the parent transfers, also one instance of REV$\langle S \rangle$ will be transferred. REV$\langle S \rangle$ can be used for interfaces between streamlets that work on a request-response basis.

NEW$\langle S \rangle$ is used to create a new physical stream that has the same dimensionality as its par-

ent, and is implicitly at the root of all streamspace types.

In Figure 2, we demonstrate by example how data structures can be mapped into streamspace.

STANDARD CONTAINER LIBRARY

As described in the Background section, software projects provide programmers with predefined containers to map data structures to memory. Containers are aliases for combinations of types from the programming language's type system, with some specific access behavior, typically implemented in a standard library. Similarly, Tydi proposes 'containers' for streamspace to represent common data structures. These 'containers' have access behavior associated with them as described by the streamspace type system. Some of these proposed mappings can be found in Table 3. The reader is encouraged to draw out some of these similar to the graphs of Figure 2, to verify the intuitive hardware-oriented view on data types of the streamspace type system.

PHYSICAL STREAMS

We discussed the streamspace types, and how it determines two properties of physical streams; E, the number of element bits, and D, the number of dimensionality bits to signal the end of a (nested) sequence. We now introduce the bit-level layout of a physical stream and show additional properties of physical streams that are relevant in the context of connecting two streamlet *interfaces* producing and consuming data. When all properties are known, a concrete circuit-level interface can be synthesized.

Physical streams have three additional properties; N, U and C. N is the number of **elements per transfer**. Communicating multiple elements per transfer can be used to scale up the bandwidth of a physical stream at the cost off additional wires. When $N > 1$, the stream has multiple **lanes** over which elements are transported. U is the number of arbitrary **user** bits piggybacking transfers, for whatever purpose. C is the **complexity level** of a stream, that describes the guarantees about the packing of elements into (mainly the temporal dimension of) streamspace. The complexity level can be used to make additional trade-offs about the complexity of the control logic of the interface on both ends of the stream, with minor nuances in area and throughput. Finally, physical streams use the same valid/ready-handshaking mechanism as AXI4-Stream for flow control.

Using these properties, the layout of a physical stream can be seen in Figure 3a. The signals fall into the following five categories.

- Flow control; the valid/ready signals for an AXI-like handshake.
- Elementary data; the N elements of size E to be transported in a single transfer, each over their own lane.
- Transfer metadata; used when $N > 1$ to deal with sub-normal transfers (i.e. when not all lanes contain valid data, explained below).
- Dimensional data; `last`, the D-bits to signal the elements are last in some dimension, and `empty`, to signal empty sequences.
- User data; `user`, an arbitrary-size field for custom per-transfer information.

In Figure 3b, we also find how the complexity parameter affects the guarantees that may be dropped when increasing the complexity level, effectively changing the number of required signals.

At the lowest complexity level $C = 1$, the source provides the strictest guarantees about the packing of the elements into streamspace. When $N > 1$, a transfer may contain less than N elements (e.g. at the end of a sequence). Requiring elements to be aligned to the least significant lane, the `end index` field signals which lane holds the last valid element. At $C >= 5$, the alignment requirement is relaxed, allowing also a consecutive number of least significant lanes to be invalid, requiring the `start index` as well. At $C >= 6$, any lane may contain valid or invalid elements, introducing the need for a `strobe`. Note that tools using Tydi can automatically insert small combinatorial conversion units in case a sink supports a higher complexity level than a source, to convert the end and start index to strobes. Note that the choice between $C = 5$ and $C = 6$ is rather significant, since when elements are very small but a high throughput is required, strobes require N signals rather than only $2 \cdot \lceil log_2 N \rceil$ signals for the end and start index. Finally, at $C >= 7$, it is furthermore allowed that every element is the last element of a sequence. In other words, a transfer may

Table 3. Overview of Tydi 'container' types.

Data type	Tydi container	Definition
Empty	Null	Bits⟨0⟩ (this is useful increase the tag size for Variant with an Empty type)
Prim⟨B⟩	Bits⟨B⟩	Bits⟨B⟩
Struct⟨T_1, T_2, ..., T_n⟩	ConcatStruct⟨$S_1, S_2, ..., S_n$⟩	Group⟨$S_1, S_2, ..., S_n$⟩
	DesyncStruct⟨$S_1, S_2, ..., S_n$⟩	Group⟨New⟨S_1⟩, New⟨S_2⟩, ..., New⟨S_n⟩⟩
Variant⟨T_1, T_2, ..., T_n⟩	PackedVariant⟨$S_1, S_2, ..., S_n$⟩	Group⟨Bits⟨$\lceil \log_2 n \rceil$⟩, Union⟨$S_1, S_2, ..., S_n$⟩⟩
	ConcatVariant⟨$S_1, S_2, ..., S_n$⟩	Group⟨Bits⟨$\lceil \log_2 n \rceil$⟩, Group⟨$S_1, S_2, ..., S_n$⟩⟩
	DesyncVariant⟨$S_1, S_2, ..., S_n$⟩	Group⟨Bits⟨$\lceil \log_2 n \rceil$⟩, New⟨S_1⟩, New⟨S_2⟩, ..., New⟨S_n⟩⟩
Tup⟨n, T⟩	ConcatArray⟨n, S⟩	Group⟨$U_1, U_2, ..., U_n$⟩, $\forall u \in U, u : S$
	Array⟨n, S⟩	New⟨S⟩
	RATElem⟨n, S⟩	Group⟨Rev⟨Bits⟨$\lceil log_2 n \rceil$⟩⟩, S⟩
	RATSlice⟨n, S⟩	Group⟨Rev⟨Group⟨Bits⟨$\lceil log_2 n \rceil$⟩, Bits⟨$\lceil log_2 n \rceil$⟩⟩⟩, New⟨S⟩⟩
Seq⟨T⟩	List⟨S⟩	Dim⟨S⟩
	Vector⟨S⟩	Group⟨Bits⟨L⟩, New⟨S⟩⟩
	RASElem⟨S⟩	Group⟨Bits⟨L⟩, Rev⟨Group⟨Bits⟨I⟩⟩, S⟩⟩
	RASSlice⟨S⟩	Group⟨Bits⟨L⟩, Rev⟨Group⟨Bits⟨$\lceil log_2 n \rceil$⟩, Bits⟨$\lceil log_2 n \rceil$⟩⟩⟩, New⟨S⟩⟩

L is a system-wide constant representing the number of bits to represent indices. RAS stands for random-access-sequence, and RAT for random-access-tuple.

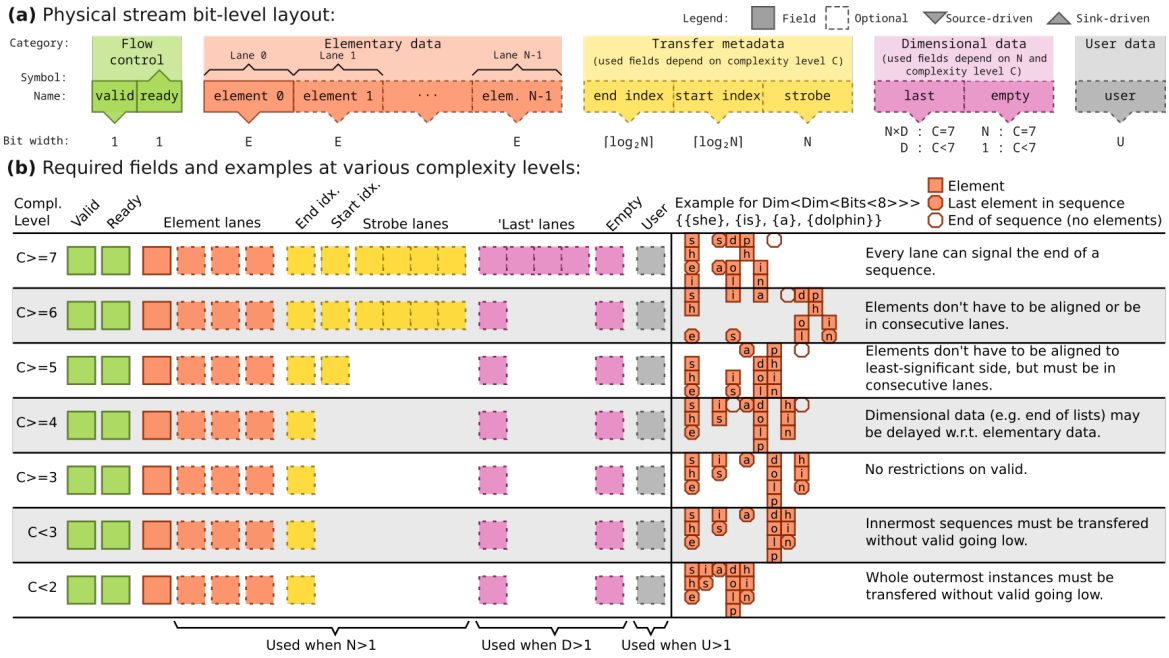


Figure 3. Bit-level layout of a physical stream (a) and examples for various complexity levels (b).

signal multiple ends of data in some dimensions, and signal multiple empty sequences. Therefore, the `last` and `empty` fields are duplicated for all lanes, linearly increasing the number of wires required for the dimensional data with respect to the number of lanes.

For a detailed discussion, we refer the reader to the Tydi website where the specification is freely available [2].

FEATURE COMPARISON

We compare the features of Tydi and existing streaming interface specifications and language abstractions mentioned in the background section. The comparison is shown in Table 4. We focus on those features that are novel through this work or common among multiple specifications.

The main difference between Tydi and AXI/Avalon is that Tydi also provides a type system for compound types (e.g. structs and variants) and describes how streams nested within streams must behave, while AXI and Avalon only describe the Tydi equivalent of a single physical stream of primitives or sequences of one dimension. While the Tydi type, and the knowledge that group and union fields adhere strict ordering clearly specifies the interaction, any logical interface with multiple physical streams using AXI or Avalon requires additional specifications.

Transferring higher dimensional information

Table 4. Feature comparison of Tydi with existing streaming interface specifications and language constructs

Feature	Specification / language construct			
	Tydi	**AXI** [3]	**Avalon** [4]	**HDLs** [6][7][8]
Intended for	Complex datastr.	Byte packets	Packets, DSP	Handshake only
Elem. size (bits)	$\{1, \infty\}$	8	$\{1, 512\}$	$\{1, \infty\}$
Structs	Yes	n.d.	n.d.	Yes
Variants	Yes	n.d.	n.d.	Yes
Stream nesting	Yes	n.d.	n.d.	n.d.
Max. dimensions	∞	1	1	n.d.
Max. data bits per transfer	∞	1024	4096	∞
Container library	Yes	n.d.	n.d.	n.d.
Multiple elem. per transfer	Yes	Yes	Yes	n.d.
Lane control	Aligned, Strobes	Strobes	Aligned	n.d.
Null elements	Yes	Yes	Yes	n.d.
Positional elements	n.d.†	Yes	n.d.	n.d.
Back-pressure	Optional	Optional	Optional	Mandatory
Multiplexing	n.d.‡	Yes	Yes	n.d.
Credit-based flow control	n.d.‡	n.d.	Yes	n.d.
User data; per ...	Transfer	Transfer	Element, Packet	n.d.

Yes: possible, by specification. **No**: not possible, by specification.
n.d.: not described by specification or documentation.
† Can be supported by using GROUP with a "don't care" bit field.
‡ Can be supported with the `user` field.

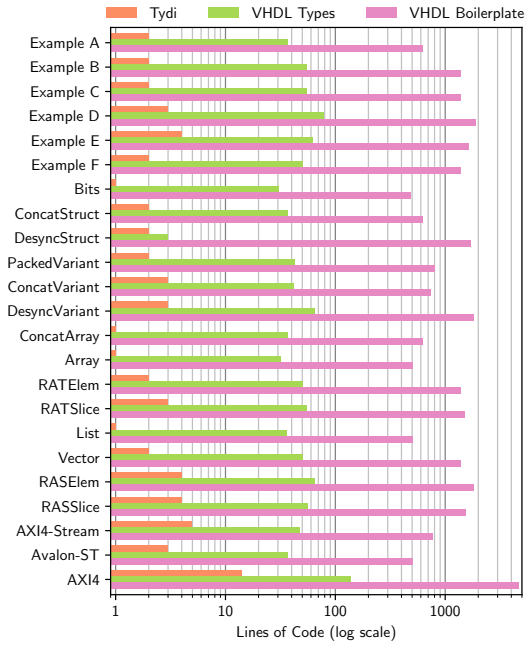


Figure 4. Comparison of hardware description effort

is also undescribed, requiring custom design effort. AXI has a unique feature, called positional bytes that a consumer should not replace in an implied byte-addressable memory being overwritten. This is an implication that is explicitly not used in Tydi, but could simply be supported by wrapping the element in a GROUP with an additional positional flag bit. AXI and Avalon contain different specific features for element packing, that are both supported through the complexity parameter of physical streams in Tydi. Avalon and AXI contain additional flow control and routing features not described in Tydi, but they can be mapped onto the `user` field.

The comparison between Tydi and the HDL constructs is rather simple, since in all HDLs that we compare, the only thing that is described and abstracted is the valid/ready handshake mechanism. Every other signal of the interface is completely user-defined. While this results in a lot of undescribed features, it provides a starting point for implementations of Tydi in the respective languages.

IMPLEMENTATIONS

We implemented a software utility, found alongside the specification, that serves as a reference implementation. The utility parses files containing declarations of Tydi types as well as streamlets with Tydi interfaces and generates HDL code templates.

Using the templates, users can build libraries of reusable components that have interfaces adhering to the specification. The back-end of the utility is modular, currently generating VHDL, but can be easily extended to other hardware description languages. The generated code consists of a package that contains user-friendly, human-readable VHDL record type hierarchies and readable boilerplate procedures derived from the Tydi types, subjectively not different from how an experienced hardware developer would write them. The generated code can be used to e.g. perform handshakes and decode unions with a single line of VHDL.

To indicate the amount of effort saved by this utility and the Tydi specification and type system, Figure 4 compares the size of the input of our utility to its output . A minimum amount of VHDL required are the record type hierarchies, shown in the figure as "VHDL Types" whereas additional boilerplate code is listed as "VHDL Boilerplate". It is design-dependent how much of this boilerplate code will be used, depending

on the procedures and functions used, so this measure gives an upper bound.

We generate code for all types presented in the examples and the container library. Only the BITS generic type parameter is used, only two fields for the containers for STRUCT and VARIANT are provided. As Table 4 shows, all the other known specifications can be implemented as a Tydi type, which we also did for the whole AXI4 (memory) interface specification. The Tydi equivalent to the HDL constructs is the Tydi BITS type.

Because the code size depends on the physical stream parameters, we generate for $E = 1$, $E > 1$ and all possible values for C, and report the average lines of code for each type. From Figure 4, we find that Tydi decreases the required lines of code of all types by an order of magnitude and potentially by another order of magnitude depending on how much of the boilerplate code is used.

We expect to implement additional back-ends for more modern HDLs, such as Chisel and Cλash in the near term. Longer term, the utility can be grown into an HDL of its own to support structural composition of streamlets, followed by behavioral constructs, where the specific rules related to the streamspace type system may be statically or dynamically checked by automated tools. Such a language could borrow from well-studied dataflow languages [10] and from recent implementations of this paradigm [11].

A subset of Tydi is also implemented in the Fletcher FPGA accelerator framework. Fletcher provides a hardware/software interface between data structures in memory and hardware accelerators. Fletcher is built on Apache Arrow, a project that provides a common in-memory data layer for over eleven software languages, preventing the need to serialize/deserialize information between heterogeneous (software) processes, which can incur significant bottlenecks in accelerator systems [9]. Because the data structures that can be expressed in Arrow include nested sequences and variants, existing streaming specifications are not adequate to support all Arrow data types, hence the need for the more advanced streaming specification and infrastructure that Tydi provides. Fletcher translates Arrow types into a subset of Tydi types, and generates the appropriate bus infrastructure and control logic to stream in Arrow data, bridging the gap between hardware and software for any of the languages supported by Arrow.

CONCLUSION

While hardware accelerators are becoming increasingly popular, we observed a lack of clear specifications and methods that allow developers to work with complex, dynamically-sized data structures in hardware description languages. We have introduced the Tydi specification, that allows to rapidly express how such structures can be exchanged between components using streaming interfaces, based on an intuitive, hardware-oriented type system. We have shown that by describing components with interfaces based on the type system, the hardware description effort can be reduced by orders of magnitude. Our work enables future integration of the type system into modern existing, or new, hardware description languages, such that the exchange of complex, dynamically-sized data structures between components is as easy to describe for hardware as they are for software today.

REFERENCES

1. L. Truong and P. Hanrahan, "A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity", 3rd Summit on Advances in Programming Languages (SNAPL 2019), Dagstuhl, Germany, pp. 7:1–7:21.
2. Accelerated Big Data System Group, Delft University of Technology. "Tydi: an open specification for complex data structures over hardware streams" [Online], January 2020. Available: https://abs-tudelft.github.io/tydi
3. Arm Limited, "AMBA 4 AXI4-Stream Protocol Specification Version 1.0", [Online], 2010, Available: https://developer.arm.com/docs/ihi0051/latest
4. Intel Corporation, "Avalon Interface Specifications", [Online], 2020
5. G. Weisz and J. C. Hoe, "CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing", 2015 25th International Conference on Field Programmable Logic and Applications (FPL), London, 2015, pp. 1-8. doi: 10.1109/FPL.2015.7294017
6. C. Baaij, M. Kooijman, J. Kuper, A. Boeijink and M. Gerards, "CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell," 2010 13th Euromicro Conference on Digital System Design: Architec-

tures, Methods and Tools, Lille, 2010, pp. 714-721. doi: 10.1109/DSD.2010.21

7. J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language", DAC Design Automation Conference 2012, San Francisco, CA, 2012, pp. 1212-1221. doi: 10.1145/2228360.2228584
8. D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis and K. Olukotun, "Spatial: A Language and Compiler for Application Accelerators". ACM SIGPLAN Notices (journal). June 2018. New York, NY, USA. doi: 10.1145/3296979.3192379
9. J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars and H. P. Hofstee, "Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow", 29th International Conference on Field Programmable Logic and Applications (FPL), Barcelona, Spain, 2019, pp. 270-277.
10. W. Thies , M. Karczmarek, S. Amarasinghe, "StreamIt: A Language for Streaming Applications", Compiler Construction, Lecture Notes in Computer Science, vol 2304, Springer, Berlin, Heidelberg (2002)
11. J. Thomas, P. Hanrahan, and M. Zaharia, "Fleet: A Framework for Massively Parallel Streaming on FPGAs." Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), New York, NY, USA, 2020, pp. 639–651. DOI:https://doi-org.tudelft.idm.oclc.org/10.1145/3373376.3378495

Johan Peltenburg is a PhD candidate in the Accelerated Big Data Systems group of the TU Delft. Johan focuses on FPGA accelerators for big data applications, working on the Fletcher accelerator framework.

Jeroen van Straten is a research engineer at the Quantum & Computer Engineering department of the Tu Delft. Jeroen works on tools for quantum simulators and digital circuit design.

Matthijs Brobbel is a research engineer at at the Quantum & Computer Engineering department of the TU Delft. Matthijs focuses on the integration of hardware accelerators in big data systems and the surrounding development operations.

Zaid Al-Ars is an associate professor at TU Delft, focusing on computing infrastructures for efficient processing of big data, and co-founder of Bluebee, a company specialized in high-performance genomics. Zaid worked in various roles in companies such as Siemens and Infineon, and serves on the advisory board of a number of high-tech startups.

H. Peter Hofstee is a distinguished research staff member at IBM and part-time professor at TU Delft, best known for his contributions to heterogeneous computing as chief architect of the Synergistic Processor Elements in the Cell Broadband Engine used in the PlayStation 3, and the first supercomputer to reach sustained petaflop operation. He currently focuses on optimizing the system roadmap for big data, analytics, and cloud, including the use of accelerated computation. Recent contributions include coherently attached reconfigurable acceleration on POWER7, paving the way for the new coherent attach processor interface on POWER8 and POWER9. He holds more than 100 issued patents.