A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems

Lukas Weber*, Lukas Sommer*, Leonardo Solis-Vasquez*, Tobias Vinçon†, Christian Knödler†, Arthur Bernhardt†,
Ilia Petrov†, Andreas Koch*
*Embedded Systems and Applications Group, TU Darmstadt, Germany
†DBlab, Reutlingen University, Germany
*[surname]@esa.tu-darmstadt.de, †[firstname].[surname]@reutlingen-university.de

Abstract—**Near-Data Processing is a promising approach to overcome the limitations of slow I/O interfaces in the quest to analyze the ever-growing amount of data stored in database systems. Next to CPUs, FPGAs will play an important role for the realization of functional units operating close to data stored in non-volatile memories such as Flash.**

It is essential that the NDP-device understands formats and layouts of the persistent data, to perform operations in-situ. To this end, carefully optimized format parsers and layout accessors are needed. However, designing such FPGA-based Near-Data Processing accelerators requires significant effort and expertise. To make FPGA-based Near-Data Processing accessible to non-FPGA experts, we will present a framework for the automatic generation of FPGA-based accelerators capable of data filtering and transformation for key-value stores based on simple data-format specifications.

The evaluation shows that our framework is able to generate accelerators that are almost identical in performance compared to the manually optimized designs of prior work, while requiring little to no FPGA-specific knowledge and additionally providing improved flexibility and more powerful functionality.

Index Terms—**FPGA, Near-Data Processing, Automatic Generation, Key-Value Store, Database, COSMOS, OpenSSD**

I. Introduction

The rate at which new data is produced and stored every day has constantly been increasing in recent years, and with the advent of the internet-of-things (IoT), this trend will continue in the foreseeable future. A substantial amount of the data produced every day is stored in database systems, such as key-value stores (KV-store). Of course, this data is not write-only: To make sense (and gain value) out of the stored data, it needs to be analyzed, ever more so now in the golden age of Big Data and Machine Learning.

Data analytics has been limited by slow I/O interfaces to the attached storage devices such as non-volatile memory (NVM). This severely hampered the processing of stored data. An interesting approach to overcome this limitation is *Near-Data Processing* (NDP): Instead of moving huge amounts of data from storage via the I/O-bottleneck to the CPU for analysis, which will eventually yield a result typically much smaller in size than the input data, Near-Data Processing places the computation much closer to the data. With hardware vendors being able to economically *integrate* processing units with non-volatile memories on a single chip or board, Near-Data Processing can help to overcome the limitations on data analytics imposed by slow I/O interfaces.

One example for a Near-Data Processing system for key-value stores was presented by Vinçon et al. in [1], [2]: By combining what they refer to as *Native Computational Storage*, which removes unnecessary abstraction layers and unifies information about data format and layout in a single layer with NDP capabilities, they were able to demonstrate speedups of up-to factor 2.7x for real-world data analysis. For their approach, they did not only use standard CPUs, but also leveraged the computational power and parallelism of FPGAs. However, the FPGA-based NDP processing elements (PEs) in their work were hand-crafted, requiring significant development effort and expertise.

In addition, not only do data storage formats *evolve* over time, but the specific data representation requirements in the actual NDP-operations also tend to change over time. Hand-crafting highly optimized NDP-accelerators becomes impractical in such scenarios, which may include data analytics on big data sets, or evolving feature vectors in machine learning.

In this work, based on the nKV architecture [1], we present a framework to *automatically* generate FPGA-based NDP accelerators from data format specifications. The generated PEs are able to filter and transform data from key-value stores, based on user-specified filter predicates and transformation rules. The PEs are integrated in a system-on-chip (SoC) architecture for the Cosmos+ OpenSSD platform [3].

In the evaluation, we compare the performance of the automatically generated accelerators with hand-crafted designs and assess the impact of the data format on the hardware footprint of the generated accelerators.

II. Motivation

In general, the development of hardware accelerators for specific applications is a tedious task that requires knowledge of the application domain, as well as expertise in accelerator development and device specifics. Typically, using the database specification, a corresponding hardware accelerator will be implemented using some form of Hardware Description Language (HDL) such as Verilog or VHDL. As soon as the accelerator design is finished, a suitable software interface has to be implemented. Depending on the target platform, this interface may vary. For the OpenSSD Cosmos+, the HW/SW interface has to be developed as device firmware, which is executed on the ARM-cores of the device. Since the architecture and the accelerator design impact how the accelerator is controlled, it is necessary to consider both when developing the software interface. As soon as the software interface is implemented, all of the components can be integrated. In this stage, the firmware is adapted to use the software interface to access the accelerator. Lastly, the hardware design (including the accelerator) has to be synthesized into a bitstream, which is used to program the FPGA-portion of the Zynq-7000 SoC on the Cosmos+. After compilation and synthesis has finished, the accelerated system can be deployed and used.

A major problem of this toolflow is the required cross-domain knowledge. Especially the PE development requires experience with hardware development, as well as a good knowledge of the target platform. Additionally, HW-SW dependencies exist, which makes it impractical to develop the software interface without a finalized accelerator design.

In this work, we aim to implement a framework which allows the *automatic generation* of the accelerator design by composing fixed architecture templates. These templates allow for the concurrent generation of the software interface. The merit of this approach is twofold: First, hardware development expertise is no longer required. The proposed framework is usable without any knowledge about hardware development or HDLs. Instead, the required information is provided to the tools in a simple C-style syntax. Additionally, the dependency between the accelerator design and the interface development is removed, allowing an overall faster development cycle.

III. Near-Data Processing Background

A. Background: Key-Value Stores

In this work, we focus on Near-Data Processing for widespread, high-performance Key-Value (KV) Stores, in particular RocksDB [4]. In order to provide querying capabilities in combination with high sustained insert and update rates, modern KV-Stores often use out-of-place update approaches such as Log-Structured Merge-Trees (LSM-Tree) [5].

An LSM-tree employs multiple components $C_0 ... C_k$. All insertions and updates are performed on the first component C_0, typically located in memory. After C_0 reaches a defined size threshold, its content, i.e., the insertions and updates, is flushed to persistent memory and merged with component C_1. Over time, the merges will gradually move data from C_0 to C_k to ensure a separation of hot and cold data. During each merge process, outdated key-value pairs are purged and their space is reclaimed.

RocksDB uses LSM trees in a *multi-leveled* variant [6]. The component C_0 comprises multiple MemTables and is located in *volatile* memory, while the remaining components $C_1 ... C_k$ reside in *persistent* memory (e.g., Flash). Whereas the MemTables in C_0 are typically implemented using a memory-efficient structure such as skip-lists, the data is transformed into the so-called *Sorted String Tables* (SST) format during the flush from C_0 to the persistent component C_1. Each component $C_1 ... C_k$ in persistent memory comprises multiple SSTs. Each SST in turn is composed by an *index block* and a number of *data blocks*. The key-value pairs are stored in the data-blocks in key-sorted order.

During the merge process, as part of the LSM tree algorithm, the SSTs are *compacted*, i.e., outdated entries are pruned. Nevertheless, as the compaction process only happens as part of the merge process, multiple key-value pairs for the same key can be present on different levels of the LSM tree hierarchy. For example, a more recent key-value pair k, v' in component C_2 supersedes a pair k, v in component C_5. For performance, no compaction takes place during the flush from component C_0 to component C_1.

Access operations to the key-value store, such as GET or SCAN require to traverse multiple index blocks, starting at the MemTables in component C_0. Assuming that the key is not present there, *all* index blocks of every SST from C_1 need to be traversed (remember that no compaction is performed during the flush, thus multiple pairs for a key can be present in C_1), followed by traversing a single index block in the remaining index components $C_2 ... C_k$. $SCAN$ operations with a value predicate (e.g., $SCAN(0 < Value < 42)$) even require traversal of the *entire* data-set.

The NDP PEs generated by our tool-flow operate on SST files using parallelized NDP operations for faster access.

B. nKV: Near-Data Processing Architecture

The NDP PEs developed in this work is based on the nKV Near-Data Processing architecture developed by Vinçon et al. [1]. Their architecture and custom key-value store exploits two key insights: First, while intermediate layers and abstraction such as block devices and file systems simplify the architecture and implementation of key-value stores such as RocksDB, they also introduce inefficiencies and complicate the implementation of true near-data processing. For NDP to be effective, it needs to operate directly on the physical addresses of the data in the key-value store. Therefore, nKV uses native computational storage, i.e., the intermediate abstraction layers along the critical I/O path have been removed and nKV directly operates on Flash storage, using physical addresses.

Having control over the physical placement of data also allows nKV to optimize the *placement* of data. By distributing data on independent Flash channels and LUNs, nKV facilitates parallel access and processing of data [1]. Moreover, keeping the data of different LSM-tree index components separated on different Flash chips, avoids blocking of the entire bus by compaction jobs taking place as part of the LSM-tree merge.

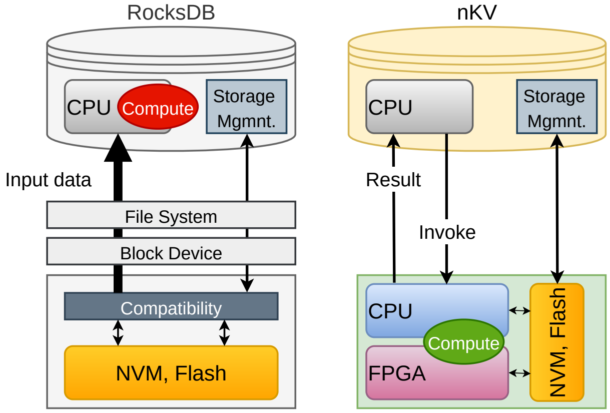

Fig. 1. Comparison of traditional KV-store and the nKV-architecture with native computational storage and Near-Data Processing.

The second important insight that underlies nKV is the fact that placing the computation closer to the data can significantly reduce the amount of data transferred, and consequently speed-up access. Many KV-store operations, such as the SCAN-operation on value predicates explained in the previous section, are very I/O-intensive, requiring much more data to be moved from storage to the processor than what is required for the final result of the operation. Using Near-Data Processing, i.e., placing the computation much closer to the data, does not only reduce the I/O complexity of the operations, but also allows for higher degrees of parallelism, as the device-internal bandwidth of storage devices (e.g., parallel access to different Flash channels) is typically much higher than the bandwidth of the I/O interface to the processor. In a similar fashion, NDP also achieves much shorter latencies.

In general, KV-Stores employ concrete data formats defined by either the application on by the database object itself (e.g. table), when applied as a DBMS storage engine, the data catalog. The nKV architecture exploits on-device data access and allows for data format interpretation in-situ. While information about the layout and format of data is scattered and encapsulated across multiple abstraction layers in classical KV-stores, nKV removes these layers and introduce on-device infrastructure for data format parsers and accessors in both soft- and hardware. The infrastructure operates on the SST format and allows interpretation of the data format and data access *without host intervention*.

The difference between the nKV architecture, with its native computational storage and use of NDP, and traditional KV-store setups, such as RocksDB, can be seen in Fig. 1: While RocksDB has to retrieve large amounts of data from the storage through intermediate layers to perform the requested operation on the host CPU, the nKV architecture can leverage the full device-internal bandwidth of the Flash and perform the requested operation on-device, eventually transferring only the much smaller result set back to the host.

While the prior FPGA-based NDP PEs for nKV were designed manually, this work will target the existing nKV architecture, and provide an *automated* tool-flow for generating FPGA-based hardware accelerators for NDP operations.

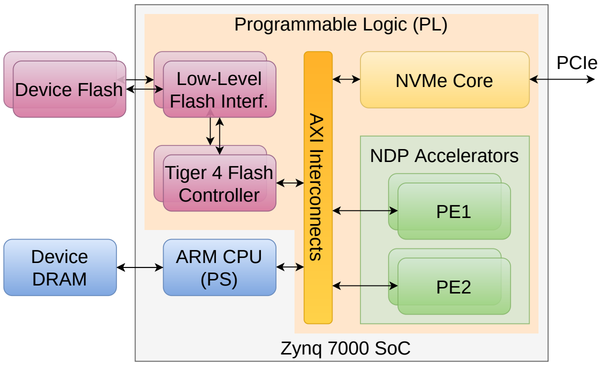

Fig. 2. Overall system architecture based on the Cosmos+ OpenSSD platform, extend with FPGA-based NDP accelerators.

IV. Near-Data Processing Accelerator Generation

Our implementation targets the Cosmos+ OpenSSD platform [3], which features a Xilinx Zynq-7000 SoC (XC7Z045). Additionally, the Cosmos+ offers two kinds of memory: Fast but volatile DRAM, and slow but persistent Flash memory.

The Cosmos+ baseline architecture enables it to be used as a “plain” NVMe SSD. To this end, the programmable logic (PL) of the Zynq SoC is used to implement an NVMe interface as well as controllers for the the attached Flash memory. Specifically, the Tiger4 Flash memory controller is used [3]. This baseline architecture is extended with FPGA-based NDP processing elements (PEs) in [1], which supports hardware/software co-execution for NDP in conjunction with the Zynq ARM cores. While [1] uses manually developed PEs, in this work we will introduce a way to automatically generate them from abstract specifications .

When adding FPGA-based PEs, a balance between Flash parallelism and compute parallelism has to be struck, since both the Flash memory controllers and the PEs compete for FPGA resources on the reconfigurable portion of the Zynq-7000. In this work, we use a single Flash DIMM and two separate Flash controllers for the Flash memory. The resulting system architecture is shown in Fig. 2.

To reduce the implementation complexity, the PEs are not directly coupled to the Flash memory. Instead, the data is first buffered in DRAM, and the results are also initially collected in DRAM. While this might seem counter-intuitive, this detour does not have significant negative performance impact due to two issues: First, the overall Flash bandwidth achievable using two Tiger4 controllers is only about 200 MB/s. Second, most of the data will be accessed multiple times, and thus profits from being stored in faster DRAM (compared to the relatively slow Flash memory).

A. NDP Accelerator Architecture Template

While the concrete functionality of the accelerators is automatically generated to match the specified filtering and data

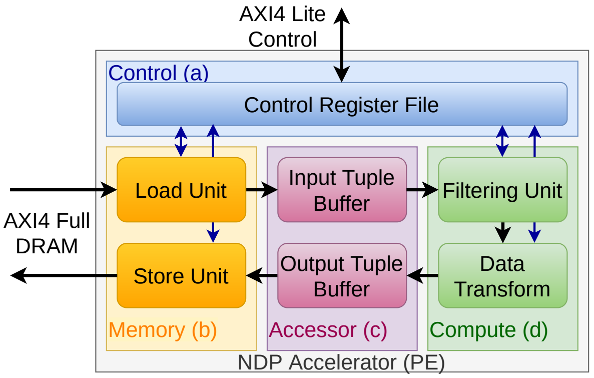

Fig. 3. Architectural template used by the generated NDP accelerators.

transformations, all accelerators use the same *architectural template* as a basis. This template, which is also depicted in Fig. 3, comprises four main components. The first component, the control component (Fig. 3.a) is simply a register file, which is mapped into the memory space of the on-chip ARM core. The registers can then be used for communication between CPU and PE.

The second component, marked (b), of the template is concerned with loading and storing data from/to memory. As described in the previous section, the PEs do not have direct access to the Flash memory. Instead, the input data is loaded from the DRAM via the corresponding AXI4 interface provided by the Zynq PS. The loading and processing of data takes place at a granularity of 32KB blocks.

The two tuple buffers in the accessor component, marked (c), are responsible for converting between the native bit-size of the memory interface (64 bit on Zynq-7000), and the actual size of a tuple in the KV-store (i.e., a key-value pair).

The computation component, marked (d) in Fig. 3, consists of two main functional units: The filtering unit will discard any tuple that does not match a user-specified predicate. Predicates can evaluate elements of the key, as well as the value and, in contrast to prior work [1], can also be defined across *multiple* columns. This is achieved by the option of chaining multiple filtering units, each evaluating a single predicate. The number of filtering stages is configurable, and the framework will automatically generate the required logic.

The second functional unit is the data transform unit, which transforms the tuples that passed the filter, as defined by the user. Example for transforms include discarding RocksDB meta-data, or unnecessary columns. Both units, the filtering unit as well as the data transformation unit, are generated automatically, as described in the next section.

B. Automatic Generation of NDP Accelerators

In general, the underlying abstraction of most contemporary databases is *structured* application data. An example for this structuring are relational databases, that impose a *database scheme* on all of the stored data. As an alternative to relational databases, key-value stores employ a less structured way of storing data. While key-value stores typically do not enforce a structure, most applications still use structured data. Thus, the application might use string-based key-value stores to store the binary data, maintaining the application-level structuring of the data outside the KV-store. The application would then use an internal record-based datatype (e.g. `structs`), and transform this data into a corresponding key-value pair. The resulting key-value pair obviously has the same structure as the underlying `struct`.

```
/* @autogen define parser Point3DTo2D with
chunksize = 32, input = Point3D, output = Point2D,
mapping = {output.x = input.y, output.y = input.z }
*/
typedef struct { uint32_t x, y, z; } Point3D;
typedef struct { uint32_t x, y;    } Point2D;
```

Fig. 4. Example Code showing how a PE is defined for automatic generation. The generated PE will automatically transform data from the `Point3D`-type to `Point2D`-type, discarding the field `x`. Additionally, the `Point3D`-structs can be filtered using predicates on all of the present fields (`x`, `y` and `z`).

For our automatic generation, we have to assume that the data is structured, as we would not be able to interpret the value data for filtering or other processing otherwise. Typically, an application will use data-classes or structs to represent this structure. By interpreting these type-definitions, our tools can generate the matching hardware NDP units for the specified data structures. In our framework, we rely on C-inspired type-definitions, as well as annotations for the specification of the PEs. This allows the database engineer to reuse his application code for the generation of PEs. An example for the specification of a PE is given in Fig. 4.

From the parsed type-definitions and annotations, an internal representation of these types is built. This internal representation is limited to data-types that are suitable for hardware-processing. Specifically, integers and single/double precision floating point types are supported. In addition to these primitive types, it is also possible to work with (nested) arrays and (nested) structs. For byte-arrays, it is also possible to flag them as string-data using a `prefix` annotation. If the annotation is given, the corresponding byte-array will be split into a `prefix` that is handled as a regular field, while the rest of the byte-array is not used for predicate-evaluation. The reason for this lies in the potential sizes of strings, which makes them very hard to process in hardware.

For example, the output of the Tuple Input Buffer is just a sequence of bits containing the complete data of the corresponding struct. With the information gathered by the contextual analysis, these bits can be interpreted. For example, consider a `struct Point` which encodes the coordinates x, y and z (all 32 bit integers) of a point in three-dimensional space. The hardware now knows, that the first 32 bits encode x, while the second 32 bits encode y, etc. Using this information, it is now possible to filter points that lie behind a certain threshold (filtering), or project the 3D-data into a two-dimensional space (data transformation).

Contextual Analysis As described previously, the contextual analysis phase of our tools is responsible for computing the data-layouts from the parsed representations of the type-definitions. To simplify this process, the contextual analysis performs multiple transformations on the struct data-type. The input to the contextual analysis are trees representing the struct-types. Each node describes a different part of the overall structs, with leaf nodes representing actual primitive types (e.g., integers), while regular nodes can be nested structs or arrays. In the first step, arrays that are annotated to represent strings are transformed into structs, which contain a `prefix`-field followed by an array, which contains the rest of the string (`postfix`). After strings are resolved in this manner, the next step removes arrays completely from the tree, by flattening them into structs with a corresponding sequence of scalar element fields. In essence, an array `uint_32t [2]` becomes the struct `struct {uint_32t elem_0, elem_1;}`. Since the data layout is identical for both, this scalarization simplifies the following steps. In a final step, the contextual analysis determines the largest *relevant* field. Relevant fields are those that can be used for filtering predicates. In our case, this includes all primitive fields *except* string-postfixes. Using the size of the largest field, the contextual analysis then determines, whether other fields have to be padded. The padding ensures that all relevant fields can be processed in a single comparator unit.

Memory Interface The memory interface contains a Load- and a Store-Unit, both having access to the PS-DRAM via a shared AXI4 Full interface. In contrast to [1], we opted for more flexible units. Vinçon et al. rely on fully static units that always load and store *complete* data blocks (32 KByte). While this keeps the hardware footprint minimal, it is not very efficient with regard to the use of memory bandwidth. Due to the Data Transformation step, which often removes elements such as metadata from the tuples, the output is almost always smaller than 32 KByte. As memory contention is a major bottleneck, reducing the number of memory accesses will improve the performance. In our work, the Load- and Store-Unit can be configured (using the Control Register File) to store variable amounts of data, thereby reducing unnecessary memory accesses and memory contention.

Tuple Buffers The Tuple Buffers transform the unstructured data retrieved from memory into processable structured data, and back again for storage. To do this, a buffer is used to group the incoming stream of 64 bit words, until one or more complete tuples are available. According to the padding and type information gathered by the contextual analysis phase, this word is split into a vector of correspondingly padded words. A second vector contains all of the disregarded string-postfixes. The string-postfixes are carried along the computations, but cannot be accessed. The Output Buffer reverses the transformation of the Input Buffer, so that the result can be stored back by the Store Unit.

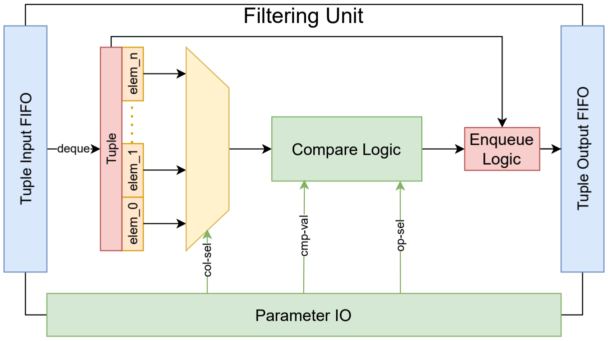

Fig. 5. Internal structure of the Filtering Unit.

Filtering Unit This module provides the selection-functionality on the incoming stream of tuples. To do this, hardware is generated that allows the comparison of tuple-members against a given reference-value using a set of compare-operations. An important extension over the work presented in [1] is the fact that the set of operators can be easily extended in our toolflow. Each operation is represented using a function mapping two data-words to a boolean value, which in turn is used to determine, whether a tuple is filtered out. Using a user-defined set of operations or the pre-defined standard set of operations ($\neq$, $==$, $>$, $>=$, $<$, $<=$, nop), the Compare Unit is generated. Since our toolflow relies on the Chisel3-framework [7] for the implementation of the actual hardware, this also enables flexibility. For example, the framework supports interfacing to Verilog and VHDL, which in turn allows addition of custom compare-operations. A schematic view of the filtering unit is shown in Fig. 5.

The input and output are FIFOs. In each cycle, a present tuple is dequeued from the input FIFO and one of its elements is selected using a multiplexer. This element is used as input to the Compare Unit which also uses the $compare\_value$ and $operator\_select$ to determine the exact operation to perform. The resulting signal is used to determine, whether the current tuple is to be enqueued into the output queue. A very important advantage of this architecture is the chainability. Due to the clear interface, this unit can be chained multiple times to allow the evaluation of multiple predicates in a pipeline, which was not possible with the architecture in [1].

Data Transformation Unit The Data Transformation Unit is automatically generated from the given struct-types. Both input and output are tuple-FIFOs. During the generation of the Data Transformation Unit, the framework will automatically match each (nested) field of the output-struct to the appropriate (if any) field of the input-struct. Using this mapping of input- to output-fields, hardware will be generated that implements this transformation. In general, there are three cases: 1) When the input and output are of the same struct-type, tuples are simply passed through. 2) If the output-struct contains *only* (nested) fields that are also present in the input-struct, the mapping is automatically derived. 3) If the output struct-type contains (nested) fields that are not present on the input, the user has to specify which (nested) input-field is to be used. While this is very flexible, it also requires user interaction in the form of corresponding annotations. An example for this is

```
/** Control Register Addresses. */
#define START 0
#define BUSY 4
[...]
#define FILTER_OP_0 60
#define CYCLE_COUNTER 64
/** Generated  Functions */
uint32_t filter_sync(...) {...}
uint32_t filter_async(...) {...}
void wait_until_done(...) {...}
```

Fig. 6. Snippet from the generated software-interface that can be used to interact with the PEs.

shown in Fig. 4 with the `mapping`-key. Using this key, it is defined that y and z are used for the projection into 2-d space. Without a mapping, the toolflow would default to the second case and use x and y for the projection.

Composition All of the described modules are then composed into a PE. Due to their latency-insensitive design, the corresponding interfaces can be directly wired-up. Additionally, all modules are automatically connected to their respective control registers. The control register file is automatically configured to provide the required number of registers.

C. Automatic Generation of the Software Interface

In addition to automatically generating the PEs for performing the NDP operations, we also added a tool pass, which automatically generates a *software-interface* for controlling the PEs. The reasoning behind this is to allow a database-engineer to use the PEs without any additional knowledge about how they work and how they are controlled.

Using the information about the Control Register File and the behavior of the PEs, we generate the software-interface bottom-up: First, we generate compiler-macros for encoding the different addresses. From these macros, we built simple software-functions for accessing the different control registers. In a final step, we use these access-functions to built more complex functionality, such as synchronous and asynchronous filtering functions using one or multiple of the filtering stages. For debugging-purposes, functions are generated for printing the state of the PE and for outputting the corresponding datatypes. All generated functions are collected in a single header-only library file, which can then be added to the project by the database-engineer in order to exploit the PEs.

An example-snippet of the generated header-only library file is given in Fig. 6.

V. Evaluation

We will first compare our automatically generated PEs against the hand-crafted units used in [1]. Since [1] has already shown that the NDP approach outperforms the typical non-NDP approach, we will omit this discussion. Then, we will examine the hardware utilizations of the generated PEs and determine their usability on the OpenSSD Cosmos+ SSD platform. All hardware-syntheses are run targeting the Xilinx Zynq-7000 SoC (XC7Z045). In all designs, the Flash controllers and processing elements are clocked at a frequency of 100 MHz, while the NVMe-Core is clocked at 250 MHz, which is in line with the original baseline. While a higher frequency could improve the performance of the PEs, the main bottleneck in this architecture is the available Flash bandwidth.

Performance For the performance evaluation, we use the same benchmarks as in [1]. They work on a sample dataset for a publication reference graph. The nodes of the graph are papers published in journals and conferences. The edges of the graph are references between those papers. Overall, the dataset is comprised of 3,775,161 Paper-Entries and 40,128,663 references between them. For the evaluation, we run GET- and SCAN-operations using the same software-NDP baseline as well as the adapted algorithm, which uses the corresponding PEs. Note that for both operations the execution is implemented in a hybrid way, where the software executes a very general algorithm and exploits the hardware whenever datablocks have to be filtered or transformed.

The resulting NDP-runtimes for GET are shown in Fig. 7 (a). Note that both the NDP hardware and software runtimes we report for GET are slightly slower (ca. 10%) than those given in [1]. This is due to updated firmware for the COSMOS+ board, which traded some performance for higher reliability. As described in [1], it also makes sense that the GET-operation does not profit greatly from hardware support, since it is sequential and the configuration-overhead (i.e., writing control registers) of accelerators is too high to make an overall difference. Even though, the GET-operation does not improve, the automatically generated PEs are similar in performance in comparison to the ones used by [1].

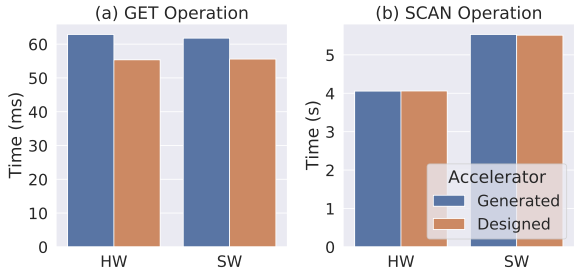

Fig. 7. Execution times of the GET and SCAN operations, comparing our work to the work provided in [1]. For both Operations execution is executed with HW-acceleration (HW) and without (SW).

The SCAN operation has much longer runtimes, making the minor firmware-induced timing variations between [1] and our measurements negligible. As in [1], the hardware-accelerated NDP SCAN is faster than the software version. The performance of our generated accelerator is on par with the manually optimized one as shown in Fig. 7 (b). Using the generated PEs slightly increases the runtime by 0.018 seconds from 5.512 seconds to 5.530 seconds.

An additional extension of our work is the possibility to generate PEs featuring multiple filtering stages. Using multiple pipelined filtering stages allows the implementation of more complex NDP-functionality. Moreover, due to the use of elastic pipelines, additional filtering stages will only add very small increases to the overall execution times. Since the filtering stages are able to process a tuple per cycle, the increase in latency of additional filtering stages will be marginal. Especially for compute-bound tasks, this would give the hardware accelerators an edge over the use of the on-device ARM-cores.

Hardware Utilization We generated accelerators that provide the same filtering and transformation functionality as [1] and compare our hardware utilization against theirs. Specifically, we use 1 paper-PE to process the nodes in the graph and 7 ref-PEs to process the edges. Since [1] only reports slices for the PEs, we limit our comparison to slices as well. Please note that each of our generated accelerators also uses a single BRAM slice, which was not the case for the custom built processing elements of [1].

TABLE I
FPGA Resource Utilization of the PEs used in [1] and our work. The design contains the complete COSMOS+ OpenSSD platform as well as 1 paper-PE and 7 ref-PEs.

	Slice Util. (abs.)		Slice Util.(%)	
	[1]	Our Work	[1]	Our Work
Overall	40821	41934	74.70	76.73
paper-PE	9480	14348	17.35	26.25
ref-PE	1277	1446	1.41	2.65
Available	54650	54650	100.00	100.00

Table I shows the corresponding utilization results. It is noteworthy that for both of the PE-types, the resource utilization has grown. Some of this can be attributed to the improved Load- & Store units, which have become more flexible. Specifically, instead of always processing blocks of a certain size, our infrastructure can be configured to load only partial data blocks. Analogously, the Store-Unit can be configured to write back partial blocks. Since the Data Transformation will typically strip data away, this reduces the overall amount of data read and written, which in turn reduces memory contention. Also, note that the overall increase is *less* than expected, considering the size increases of the individual PEs. This is due to a more efficient use of interconnects in our refined architecture template.

We also evaluated the amount of hardware required for multi-staged filtering, as well as for different tuple sizes. For the first part, we take a closer look at the correlation between tuple-sizes and required hardware. For this part of the evaluation, we rely on out-of-context synthesis. In out-of-context syntheses, only a selected part (in our case the PE) is synthesized without the rest of the surrounding architecture. The resulting utilizations represent the amount of logic resources required *without* very dense packing. For the generation of the PEs, we used a number of different input formats that feature tuple sizes ranging from 64 bits up to 1024 bits. For of these sizes, we specified a `struct` with the corresponding number of `uint32_t` and `uint8_t` values. Input and output types are identical and mapped automatically. For each size, we generate a PE that is able to compute on the complete tuple (at the granularity of 32-bit fields) and another PE, where half of the data is discarded using string-prefixes.

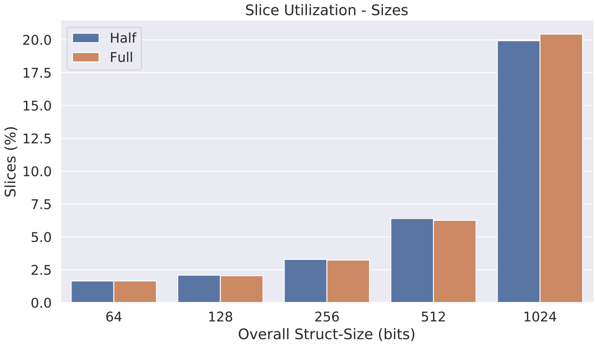

Fig. 8. Out-of-Context Slice Utilization of generated PEs in correlation to the size of the processed tuples. Half refers to accelerators using the prefixing, whereas Full refers to the ones using all data.

The results are shown in Fig. 8. An interesting observation is the fact that for smaller PEs, the use of string-prefixing yields a higher slice-requirement. At a first glance, this would make the prefixing irrelevant. To understand why prefixing is still necessary in some cases, we have to consider that the critical part of our hardware is the Filtering Unit with the compare operations at its core. In Fig. 9, all fields have a width of 32 bit, which means that the corresponding compare-operators are also 32 bit operators. For the 1024 bit struct, the corresponding string-data would have an overall size of 512 bits. A full-width compare unit would vastly increase the amount of required hardware. Thus it is still reasonable to use the prefixing.

Lastly, we take a closer look at the multi-stage feature and the resulting hardware-requirements. For this part of the evaluation, we reuse the same data-formats as in the previous step, but focusing on 256 bit structs only. For both (with and without string-prefixes), we built accelerators with up to 5 filtering stages for more complex predicates. Of these, especially the 2-staged ones are interesting, since they could be used to implement RANGE_SCANs. Again, the utilization results were obtained using out-of-context synthesis.

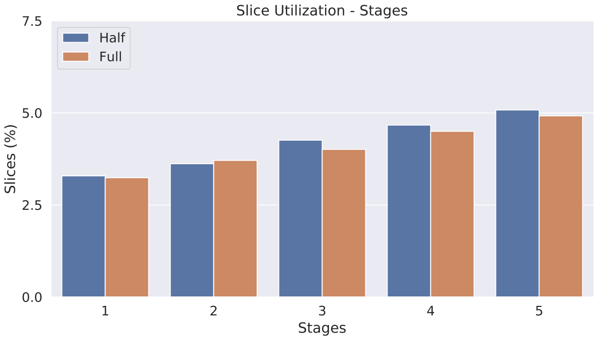

Fig. 9. Out-of-Context Slice Utilization (in percent) of generated PEs in correlation to the number of filtering stages. Additional stages increase resource requirement in a linear fashion, but provide more flexibility. The use of string-prefixing (Half) has only minor impact.

Looking at the results shown in Fig. 9, we can see an almost linear correlation between the number of stages and the slice requirement. Additionally, we observe that the increase per additional stage is small compared to the overhead incurred by the fixed part of the template (Load/Store Unit, Tuple Buffers). This implies that multi-stage filtering incurs only minor additional cost, while offering a lot more flexibility.

VI. Related Work

The first approaches for Near-Data Processing, moving computation closer to the data date back to as early as the 1970s. However, approaches such as database machines [8] or ActiveDisk [9]–[11] were severely limited by the I/O-limitations and memory bandwidth of mechanical hard-drives.

Only after the wide-spread availability of modern non-volatile storage solutions, e.g., Flash-based SSDs, significant advances in the performance of Near-Data Processing systems became possible. Approaches such as SmartSSD [12]–[14] exploit the much higher I/O-bandwidth of modern storage devices as, for example, provided by parallel, independent Flash-channels. JAFAR [15], [16] was one of the first systems focusing on Near-Data Processing for DBMS. Biscuit [17] was another approach targeting NDP for DBMS, namely MySQL. In contrast to our work, they only employed the ARM-based CPUs found in commodity SSD hardware for software-based Near-Data Processing, but also identified the lack of a usable framework for programming NDP PEs as an important issue. Our framework allows to automatically generate FPGA-based Filtering and Data Transformation units from simple user-input. It thus offers a solution to make FPGA-based NDP acceleration accessible to non-FPGA experts.

With their HRL architecture [18], Gao et al. present a new hardware architecture targeting NDP that combines fine-grained reconfigurable regions, as found on FPGAs, with coarse-grained regions as common in Coarse-Grained Reconfigurable Arrays (CGRA). Their overall system architecture combines this accelerator with DRAM in an Hybrid Memory Cube (HMC), but does not include non-volatile memories.

Architectural challenges and other considerations on how to integrate FPGAs into Near-Data Processing architectures were discussed by Dhar et al. [19] and Becher et al. [20]. While Dhar et al. envisioned an architecture featuring Flash storage and a combination of FPGA and High-Bandwidth Memory (HBM), with the FPGA processing data cached in HBM, the ReProVide architecture proposed by Becher et al. uses a combination of an ARM CPU and an FPGA, similar to our approach. In the multiple dynamically reconfigurable regions of the FPGA, different pre-synthesized NDP PEs can be used. However, these accelerators must be hand-crafted and cannot be generated automatically.

VII. Conclusion & Outlook

In this work we have developed a framework for the automatic generation of FPGA-based accelerators for the use with Near-Data Processing applications. Our evaluation shows that our automatically generated accelerators provide almost identical performance compared to a setup with hand-crafted hardware accelerators. This is worthwhile, since our approach effectively removes the need for custom hardware development and lowers the entry barrier for hardware-accelerated databases. Moreover, our multi-staged filtering approach enables more powerful computations with minimal overhead.

While filtering and transformation of data are wide-spread use-cases that can easily be realized using our framework, more computational and analytical tasks could also be performed using this architecture. In future work, we will investigate, how we can leverage the data-parallelism of the architecture to perform more compute-intensive tasks. Using our architecture, it is possible to access and process all tuple-elements in parallel, which could offer great potential for faster analysis of the processed data.

Acknowledgements

This work has been supported by BMBF PANDAS - 01IS18081C/D, DFG Grant neoDBMS - 419942270 and HAW Promotion, MWK, Baden-Würrtemberg, Germany.

References

[1] T. Vinçon *et al.*, “Nkv: Near-data processing with kv-stores on native computational storage,” in *Proc. 16th International Workshop on Data Management on New Hardware*. ACM, 2020.

[2] L. Weber *et al.*, “On the necessity of explicit cross-layer data formats in near-data processing systems,” *Distributed and Parallel Databases*, 2021.

[3] Y. H. Song, S. Jung, S.-W. Lee, and J.-S. Kim, “Cosmos+ openssd: A nvme-based open source ssd platform,” *Flash Memory Summit*, 2016.

[4] Facebook, “Rocksdb,” https://github.com/facebook/rocksdb, 2020.

[5] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” *Acta Inform.*, 1996.

[6] C. Luo and M. J. Carey, “Lsm-based storage techniques: a survey,” *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[7] J. Bachrach *et al.*, “Chisel: Constructing hardware in a scala embedded language,” in *Proc. DAC 2012*, 2012.

[8] H. Boral and D. J. DeWitt, “Parallel architectures for database systems,” 1989, ch. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28.

[9] A. Acharya, M. Uysal, and J. Saltz, “Active disks: Programming model, algorithms and evaluation,” in *Proc. ASPLOS 1998*, 1998.

[10] K. Keeton, D. A. Patterson, and J. M. Hellerstein, “A case for intelligent disks (idisks),” *SIGMOD Rec.*, 1998.

[11] E. Riedel, G. A. Gibson, and C. Faloutsos, “Active storage for large-scale data mining and multimedia,” in *Proc. VLDB 1998*, 1998.

[12] J. Do, J. Patel, D. DeWitt, and e. al, “Query processing on smart ssds: Opportunities and challenges,” in *Proc. SIGMOD 2013*, 2013.

[13] S. Seshadri, S. Swanson, and et al., “Willow: A User-Programmable SSD,” *USENIX, OSDI*, 2014.

[14] Y. Kang, Y.-s. Kee, and et al., “Enabling cost-effective data processing with smart SSD,” in *Proc MSST 2013*, may 2013.

[15] S. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, “Beyond the Wall: Near-Data Processing for Databases,” *Proc. DAMON*, 2015.

[16] O. O. Babarinsa and S. Idreos, “JAFAR : Near-Data Processing for Databases,” 2015.

[17] B. Gu *et al.*, “Biscuit: a framework for near-data processing of big data workloads,” *ACM SIGARCH Computer Architecture News*, Jun. 2016.

[18] M. Gao and C. Kozyrakis, “HRL: Efficient and flexible reconfigurable logic for near-data processing,” in *2016 IEEE Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2016.

[19] A. Dhar *et al.*, “Near-Memory and In-Storage FPGA Acceleration for Emerging Cognitive Computing Workloads,” in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Jul. 2019.

[20] A. Becher *et al.*, “Integration of FPGAs in Database Management Systems: Challenges and Opportunities,” *Datenbank-Spektrum*, vol. 18, no. 3, Nov. 2018.