2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 978-1-6654-3555-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/FCCM51124.2021.00009

XBERT: Xilinx Logical-Level Bitstream Embedded RAM Transfusion

Matthew Hofmann,* Zhiyao Tang,* Jonathan Orgill,† Jonathan Nelson,† David Glanzman,‡
Brent Nelson,† and André DeHon*
*Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
†Dept. of Electrical and Computer Engineering, Brigham Young University, Provo, UT, USA
‡Nvidia Corporation, Santa Clara, CA, USA
Email: {matth2k,zhiyaot}@seas.upenn.edu, nelson@ee.byu.edu, andre@ieee.org

***Abstract*—XBERT is an API and design toolset for zero-cost access to the on-chip SRAM blocks on Xilinx architectures using the device's configuration path. The XBERT API is high-level, allowing developers to specify DMA-like data transfers of memory contents in terms of the logical memories in the application source code and thus is applicable to essentially any design targeting Xilinx devices. XBERT is broadly accessible to application developers, hiding the low-level details of physical mapping and bitstream encoding. XBERT is efficient, consuming zero reconfigurable resources with no impact on Fmax. XBERT achieves a bandwidth of 3–14 megabytes per second (MB/s) and complete readback and translation of a memory in an isolated 36Kb block RAM in less than 0.5 ms on a Xilinx UltraScale+ MPSoC Zynq.**

I. Introduction

Today's FPGA fabrics include a large number of distributed, embedded RAM blocks. While the most common use of these embedded RAMs is to supply and store data used during computations on the FPGA fabric, it is ocassionally necessary to load them with data or inputs from outside the fabric or to retrieve data that they store as output from the computation.

We could use FPGA fabric resources and dedicated fabric input/output channels (e.g., AXI channels) to move data to and from these embedded memories, but that consumes FPGA fabric resources as well as the limited read/write ports on the embedded RAM primitives. And, this extra logic can create challenges to meet application timing requirements [1].

The contents of these embedded RAMs are also accessible through an existing, dedicated on-chip network—the bitstream reconfiguration path. Consequently, it should be possible to use the bitstream reconfiguration path to move data in and out of these embedded RAM blocks without consuming any FPGA fabric resources. Furthermore, on SoC FPGAs with embedded processors, the processor can manage reconfigurations as part of the system computation.

However, mapping *logical memories* found in the original design source to *physical memories* on the FPGA fabric is a challenging task. Once the implementation tools have mapped a large logical memory onto multiple smaller memories, identifying which physical BRAM primitive holds particular logical data is something not readily apparent to the application programmer and something that may change every time the design source is changed and remapped to the FPGA. Further, the format of data used for bitstream programming is not the same as the logical format for the stored data—the bits of a logical memory are not arranged in neatly ascending order within the BRAM initialization strings and the bits in those initialization strings are subsequently scattered across a number of configuration frames.

To address these needs, we created the Xilinx Bitstream Embedded RAM Transfusion (XBERT) API to provide a high-level interface to read and write the contents of *logical memories*—memories as seen by the application developer in the source-level (e.g., RTL, HLS, IP Blocks) design.[1]

Transfusion operations are combined read and write operations that extract old BRAM contents and supply new BRAM contents. Importantly, these combined transfuse operations can be more efficient than using separate read and write operations.

Our automatic tool flow extracts the information about how logical memories are mapped to physical embedded RAMs to allow the XBERT API to operate on those memories. Our API is able to read and translate the resulting data to a logical memory format in a fraction of a millisecond providing effective data transfer rates in the MB/s. This is not as fast as a dedicated, high-speed link (e.g., AXI channel at 5 GB/s), but is adequate for memories that are accessed infrequently (e.g., to program with unique data at startup or recover data when the program completes) or at modest bandwidth (e.g., periodic parameter adjustments) as illustrated in Secs. II and III-F. And significantly, this is achieved with no reduction in Fmax or consumption of FPGA resources.

Our contributions include:

- An API to provide *logical* access to memories stored in embedded RAMs in the FPGA fabric (Sec. V)
- An open-source implementation of the XBERT API for the UltraScale+ MPSoC Zynq, including both the tool flow to extract the logical↔physical mapping for individual memory bits and runtime support to read and write the running FPGA's memory contents (on-line at [2])
- A customized compression technique for the logical-to-physical bit mapping that reduces translation table size from megabytes to kilobytes (Sec. VII)

[1]Technically, XBERT works on the logical memory organization as reflected in a DCP (Design CheckPoint) file.

978-1-6654-3555-0/21/$31.00 ©2021 IEEE
DOI 10.1109/FCCM51124.2021.00009

- A table-based acceleration that can reduce single BRAM translation times to be comparable to DMA transfer times (Sec. VIII)
- A characterization of the performance of the API implementation (Sec. IX)

II. Customers of On-Line Transfusion

Many use cases exist for the general capability provided by XBERT. Programming BRAM-configured overlay architectures is one obvious need to infrequently modify embedded RAM contents, including updating program memory contents in embedded soft processors such as RISC-V processor cores [3], [4], custom VLIWs [5], and customized VLIW [5], [6], and Vector [7], [8] processors. This need to load instructions also applies to loading configurations for more specialized overlay fabrics such as dedicated FSM evaluators [9], [10] and Neural Networks [11] or simulators [12].

Another use case is at-speed unit testing of FPGA building blocks, where it is desired to feed the module-under-test with data at full rate and capture the results. This can be done with a memory to source data into the module-under-test and a second memory to record the results. Similarly, in live debugging with an Internal Logic Analyzer (ILA) or trace buffers [13], [14] to capture data during operation, the speed at which we offload this data after a test is often not critical.

Finally, at the extreme, XBERT functionality could provide an inexpensive way to support advanced abstractions like CoRAM [15] on current FPGAs. CoRAM proposed adding dedicated infrastructure to manage data movement between embedded RAMs and a central memory and demonstrated prototypes that built an overlay network on top of the FPGA. Using XBERT, the same functionality could be provided using the existing reconfiguration path hardware support without the cost of the added overlay logic.

III. XBERT: Overview, Challenges, and Solutions

This section introduces XBERT, including the challenges and operational requirements for a system like XBERT along with its solutions to those problems and the resulting benefits.

A. Basic Single BRAM Operation

To start out, consider the simple use case of changing the contents of a simple logical memory that maps to a single 36Kb BRAM. If the logical memory happens to be an XPM instantiated memory, Xilinx's `UpdateMem` program, running on a host, can change the BRAM in the full bitstream. It takes 4 seconds to run `UpdateMem` to change the bitstream to reflect the new BRAM contents and 9 ms to load the full bitstream onto a XCU3EG. This is both slow and demands the use of a separate host machine.

Using the simplest write API in XBERT (`bert_write`) we can perform the update while running code on an embedded APU core on the Zynq and write the BRAM in 1.04 ms (over 3900× faster). This includes 0.71 ms to translate the bits (convert the logical memory bit description into a partial bitstream) and 0.25 ms to write the partial bitstream to the device through the Zynq PCAP; the write achieves 3.9 MB/s bandwidth. Importantly, this XBERT write operation works for any logical memory, not just those instantiated with an XPM.

B. Table-Based Translation, Compression, and Acceleration

While `UpdateMem` runs using the full Xilinx device database for the part and the full set of design checkpoint data for the design, XBERT replaces this information with a minimal translation table describing where each logical memory bit is mapped into the bitstream. This raw translation table in XBERT initially takes 238 KB, but using compression we can reduce that to 2.3 KB (Sec. VII).

The dominant component of the 1.04 ms XBERT time above is in translation (0.71 ms), which consists mainly of computing the bitstream locations for the individual bits from the logical memory image. When we add an accelerated, multi-bit, table-based translation ability to XBERT (Sec. VIII), we can reduce this to 0.28 ms, so the entire write occurs in 0.62 ms for a throughput of 6.59 MB/s—about twice as fast our unaccelerated case.

C. Transfusion - Combining Operations For Performance

Turning to the minimum DMA transfer time for our single BRAM memory, we note it is large (0.25 ms), in part, because writes occur in frames, requiring the writing of data for all 12 memories that share a frame in the UltraScale+ architecture (Fig. 3). If we need to write many BRAMs in a frame, either because a logical memory uses many BRAMs, or because we need to write many logical memories that happen to share a physical frame, we can reduce the DMA transfer time per BRAM. At the extreme, we bring the per-BRAM DMA transfer cost down by a factor of 12 to around 0.021 ms, so the total per-BRAM write cost is around 0.32 ms (or a throughput of 11.75 MB/s—about twice again the single accelerated throughput). As a result, it is efficient to have a scatter-gather interface that allows us to specify the full set of logical memories we would like to write as a single operation so the API can minimize the number of frame writes required. Our transfuse API interface provides this capability (Sec. V).

While there are write enables that allow us to write a single 36Kb BRAM in a frame at a time, there are no write enables to control independently writing the two 18Kb memories within a single 36Kb BRAM. To write one 18Kb memory in isolation, we may need to read the entire frame first in order to preserve the value of a partner 18Kb memory in the block. The XBERT transfuse API can combine this readback with the read of other logical memories in the same physical frame (Sec. V).

D. Dealing With Larger Memories — Logical To Physical Memory Mapping

For simplicity, the examples above used a logical memory that mapped to only a single BRAM. In practice, logical memories often map to multiple physical BRAMs. When this happens, there are many choices for how the bits are packed into memories. For example, in one 32×10,000 memory, we have seen that Vivado mapped the bottom 18b ([17:0]) of the

memory to five 18×2048 RAMB36s, the next 9b ([26:18]) to three 9×4096 RAMB36s, the next 4b ([30:27]) to two 4×8192 RAMB36s, and the final bit ([31]) to one 1×16,384 RAMB36; this non-uniform mapping uses only 11 RAMB36s, whereas in other designs, we have seen Vivado map a similarly sized memory to a set of 16 RAMB36 memories, each 2 bits wide by 16K words deep.

Finally, for memories smaller than a single BRAM, Vivado may tie off high order address bits to values other than 0 meaning the logical 0 location for the memory does not start in the normally expected location (the first frame of the frameset for the physical BRAM).

To address these complicated and tedious physical mapping issues, the XBERT toolflow automatically extracts these mapping details from the Vivado Design CheckPoint (DCP) file or project as a part of its Logical to Physical Memory Mapping capability. As a result application developers need not deal with them—they are able to deal exclusively with logical memory contents, and the many mapping details are hidden from them by XBERT (Sec. V).

E. XBERT Design Flow Summary

Fig. 1 illustrates the host side preparation tool flow for XBERT. Starting at the top, the user's design (represented minimally in the form of a design checkpoint) is processed to extract information on the logical memories contained in the original design source, resulting in an MDD file (Sec. IV-D). That is then combined with bitstream information extracted from Xilinx-produced `.ll` files to create a complete representation of the logical-to-physical memory mapping information for the design (`mydesign_uncompressed.{c,h}`). This is then compressed and optionally accelerated to produce a `mydesign.c` file containing a compressed version of that information as C data structures.

At the bottom of the figure, a final XBERT application is assembled from three sets of source code: (1) the XBERT runtime source code (`bert.c`), (2) the `mydesign.c` file, and (3) the user's application code (`application.c`). This is linked against XBERT's extended `xilfpga` libraries (Sec. VI) into the final executable application program.

F. A Full Motivating Example

Consider developing a Huffman encoding accelerator that contains four independent memories. This Huffman encoder takes in a stream of bytes and compresses it by mapping each byte to a variable-length code. And, for good compression, the Huffman code should be tuned to the data being encoded.

We design the encoder to work for any encoding using an encoding table (memory #1) and can use XBERT to update its contents when changing encodings. The encoding table in the HDL source is read-only but we can use `bert_write` to load its contents. This takes 1.1 ms to load the table. Without XBERT, we would need to use an AXI port to perform the load, which would be faster, but adds LUTs and registers to the design (Tab. I).

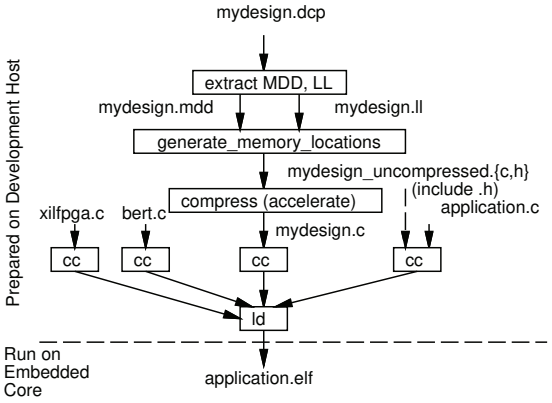


Fig. 1. XBERT Tool Flow

TABLE I
IMPACT OF ADDING AXI ACCESS TO MEMORIES IN HUFFMAN EXAMPLE

Variant	LUTs	FFs	Fmax
No AXI (XBERT only)	165	80	425 MHz
+ AXI for encoding table	1360	1125	405 MHz
+ AXI for histogram memory	2272	1818	405 MHz
+ AXI for results memory	3112	2531	405 MHz
+ AXI for input data memory	3903	3176	405 MHz

When we first design the encoder, we often want to test it for functionality and speed before the data source and data consumer are added to the design. We can do this by adding a memory to hold the input data (memory #2) and a memory to store the output data (memory #3). Using XBERT we need nothing else. We can load new data to be compressed by the encoder into the input memory using `bert_write` and recover the compressed output using `bert_read`.

We also add a histogram memory to capture the characteristics of the input data using a 256-element memory (memory #4). Using `bert_read` we can read the contents of this memory back in 0.45 ms. If we're compressing one byte per cycle, this histogram would need to perform both a read and a write on each cycle, meaning both its ports are being used. For a non-XBERT design to provide readback, it would need to share one of the ports or duplicate the memory to effectively provide another read port. With XBERT neither is required.

Using XBERT we can read or write any of the four memories in the design. We could read the histogram memory contents, compute a new encoding table tailored to the input data stream, and write the new encoding table back to the design, all in under 1.2 ms.

Tab. I shows the impact (area, Fmax) of adding AXI interfaces to the design. We use an MMCM between the AXI subsystem and the Huffman decoder so the latter can run at its maximum clock rate and not be limited by the maximum AXI clock rate of 333 MHz. Since this Huffman example is small, the Fmax impact of providing access to the BRAMs is small. In larger, highly congested designs, [1] claims eliminating AXI BRAM access can realize as much as a 63% Fmax improvement (Sec. IV-C).

IV. Related Work

A. Physical BRAM Manipulation Applications and Tools

A number of tools have exploited cases where they could provide useful functionality working with raw BRAMs without requiring mapping or translation back to the logical level. As early as 2000, Xilinx provided experimental, low-level support for partial reconfiguration readback and reloading of embedded RAMs [16], [17], used mainly for memory readback and scrubbing [18]. More recently, ReconOS uses bitstream reads and writes for multitasking [19] (to load and unload BRAM contents via bitstreams). Metawire uses bitstreams to move data from BRAM to BRAM to provide Network-on-a-Chip (NoC) functionality; since they control the BRAM mapping and move from BRAM to BRAM, they avoid the need to translate from physical to logical mappings [20]. Similarly, several works have used direct BRAM writing for specific applications including [21]–[23].

B. Physical Bitstream APIs

More recent work has aspired to provide an API for physical-level bitstream manipulation. BITMAN provides a general physical level access mechanism to BRAM contents with their `change_BRAM_content(X, Y, new_config)` API [24]. As with the above tools, it requires that some higher-level interface or manual developer intervention determine which BRAM locations need to be changed and to format the configuration data, including shuffling the logical bits to their locations in the physical configuration mapping.

Similarly, recent work in [25] advocates this approach of using bitstream readback and edits to read and write BRAM contents and demonstrates their use to copy data between BRAMs. But, it does not address identifying which physical BRAMs are used for a particular logical memory. Nor does it provide a complete description or high-level tool that will allow a developer to map their HLS or RTL logical memory contents into the bitstream or to extract bitstream contents and reconstruct the state of the HLS or RTL memory.

C. Specialized Logical Memory Manipulation

Maxeler explores using the bitstream path to load and read their "Mapped Memories" instead of a separate low-speed bus [1]. They show that removing the low-speed bus and its associated demand on fabric resources increases the performance of their designs by up to 63%. Their bitstream interface achieves up to 2 MB/s data transfer bandwidth.

The Maxeler use is, perhaps, most similar to what XBERT provides. However, Maxeler (a) only uses this BRAM path for a specific use of memories generated internally to their compiler, (b) does not provide a general API available to developers for use for any memory or any tool chain, and (c) their tool must take control itself of the mapping of logical memories to BRAMs because they do not have enough information to determine how the Xilinx tools map logical memories to physical memories.

Similarly, Xilinx Vivado provides support for changing the initial value of configuration memories in a bitstream mostly to support instruction memories for MicroBlaze processors [26]. This includes a BRAM Memory Map Infromation (MMI) file that records the physical BRAMs used to support MicroBlaze and Xilinx Parameterized Macro (XPM) memories and the `UpdateMEM` tool that can update the bitstream with data from a logical memory file [27]. The MMI generated by Vivado does not cover all logical memories in the design, and `UpdateMEM` only produces a complete bitstream.

XBERT closes all of the above gaps in these related works by providing open-source, logical-level access to the bitstream read and write path. It accommodates all designs, all memories, and all tool flows that go through DCPs.

D. Existing Bitstream Manipulation Support

A number of tools are available that can be helpful in creating bitstream manipulation tools. The XBERT system is based, in part, on some of them.

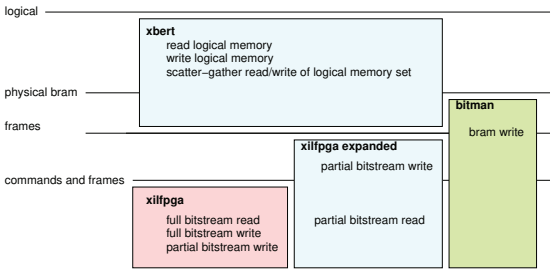
The Xilinx tools have long produced Logic Location (LL) files for memories as part of bitstream readback generation [28]. The LL file contains the frame and bit location for every data bit in a physical BRAM, but provides no mapping information about the logical memories and how a logical memory is mapped onto multiple physical BRAM tiles. Nonetheless, the LL file is useful for deciphering the frame and bit positions for a BRAM within XBERT as show in Fig. 1.

Project X-Ray [29] and the more recent Project U-Ray [30] provide databases mapping the configuration bits in Xilinx 7-series and UltraScale+ devices to frame and bit locations. These contain no information on logical-to-physical memory mapping within a specific design but do provide information and tools for bitstream encoding (understanding the locations of physical memory bits within the bitstream).

Building upon Project X-Ray, BYU developed an open-source tool, `prjxray-bram-patch` [31] that serves a similar role as Vivado's `UpdateMEM`, but works for all logical memories in a design and for all design flows. The `bram-patch` tool defines a Memory Description Data (MDD) file that plays a similar role to the Xilinx MMI file. Derived from a Vivado design using Tcl, an MDD file describes, for each logical memory in a design, the collection of physical BRAMs to which the logical memory is mapped, how the logical memory bits were partitioned between the physical memories, and how the bits are packed into the INIT strings of a physical RAMB primitive. The XBERT system uses this information as a part of its logical-to-physical mapping step (Fig. 1).

V. The XBERT API

XBERT is designed to run on the embedded APU cores on Zynq SoCs. As shown in Fig. 2, it provides a logical-level interface above that provided by `xilfpga` or the interface provided by physical-level bitstream manipulation tools such as BITMAN [24]. It does this by extending `xilfpga` with



Light red provided by Xilinx; light blue provided by this work; light green from [24].

Fig. 2. API Layering for XBERT

partial reconfiguration support and command generation for BRAM writes (Sec. VI).

The entry-level XBERT interface is a simple pair of routines providing a DMA-like interface to read or write an entire logical memory.

```
int bert_read(int logicalm,
              uint64_t *data,
              XFpga* XFpgaInstance);
int bert_write(int logicalm,
               uint64_t *data,
               XFpga* XFpgaInstance);
```

These routines take care of bit shuffling between the frame format and the logical format of the memory as well as performing the needed partial reconfiguration reads and writes.

To support these API calls, the XBERT preprocessing tools process the MDD files (Sec. IV-D) to produce files `mydesign.h` and `mydesign.c` (Fig. 1). These include C code definitions of the specific memories in the design including the logical-to-physical translation tables.

One limitation of the simple API above is that every read or write call first does a logical to physical memory translation followed by a bitstream read or write operation to the device. This can be inefficient if the application needs to read or update many different logical memories that are mapped to the same configuration frames.

To allow more efficient transfer, XBERT also provides an API that can read and write a set of multiple memories like a scatter-gather DMA operation. This is the `bert_transfuse()` call that allows XBERT to perform one set of frame reads, translations, and then a single set of frame writes covering all the logical memories that may share a set of frames:

```
int bert_transfuse(int num,
        struct bert_meminfo *info,
        XFpga* XFpgaInstance);
```

The `bert_transfuse` routine takes an array describing the operations on a collection of logical memories. To describe each transfusion operation, it uses a structure (`bert_meminfo`). The structure specifies the memory, the operation (read or write), and the logical address range we are reading or writing in the memory. To support data wider than 64b, this supports an array with multiple data words for each array slot. And, we allow operations on a subset of words in the logical memory by specifying a start address and length. This allows access and update to a subset of the frames associated with the BRAMs holding data for the logical memory, which is more efficient when only a portion of the memory needs to be read or written. The transfuse operation also arranges the set of write data along with PCAP control instructions so that they can be performed with a single DMA write transfer, minimizing the overhead of DMA setup.

The APIs assume that it is safe to perform the read and write operation. Putting the computation into a safe state where the memories are not being written during the read or write is the responsibility of the application. For example, using the standard Xilinx IP block-level interface protocol generated by Vivado/VitisHLS [32], one might watch for the block to be done (`ap_done`), perform the XBERT operations, then restart the module (`ap_start`).

VI. Partial Reconfiguration

Partial reconfiguration is the loading of configuration data for a portion of the FPGA resources without disturbing the operation of the remaining resources. In modern Xilinx devices, the atomic unit of configuration is a frame that is organized along FPGA columns. In the Xilinx UltraScale+ series, each frame has 93 32b words. Since BRAM data constitutes a small fraction of the total bitstream, using partial reconfiguration to access only BRAM frames reduces bitstream read or write time compared to full bitstream reads or writes.

In modern FPGA devices, embedded RAMs are placed in columns. The UltraScale+ series have 36Kb BRAMs (RAMB36) that can each alternately be configured as a pair of 18Kb BRAMs (RAMB18). Each frame in the Xilinx UltraScale+ series device covers 12 RAMB36 memories [33, Chapter 8, Configuration Frames] and has 144b for each RAMB36, 72b for each of the two RAMB18s. It takes 256 frames to cover the BRAM group (See Fig. 3). The set of 144b per BRAM in a frame are grouped into 240b blocks. There is a write enable bit for each 144b RAMB36 group in a frame roughly in the middle of the 240b block. The write enables allow updates of a single BRAM36 at a time, but it is always necessary to transfer data in units of frames. BRAM data frames are separate from frames that hold routing or LUT configurations.

Zynq devices include a Processor Configuration Access Port (PCAP) to allow the embedded processor to read and write configuration frames. For high-speed access, the processor can configure DMA data tranfers to the PCAP to perform partial reconfiguration operations. The UltraScale+ PCAP is 4B wide with a peak operation of 200 MHz supporting up to 800 MB/s [33, Chapter 8, Configuration Time].

Xilinx provides the `xilfpga` library [34] (Fig. 2) for performing DMA bitstream transfers on Zynq UltraScale+ components. It allows full bitstream load and readback and

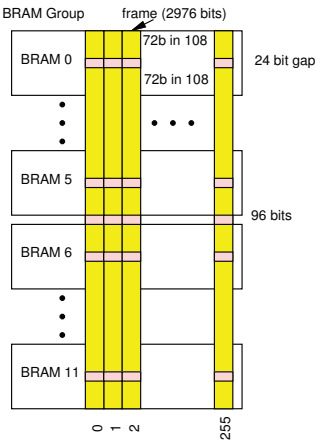


Fig. 3. Frame Organization for UltraScale+ BRAM Contents

partial bitstream load for `.bit` files that include a header of control commands to the PCAP.

XBERT provides expanded versions of routines in the `xilfpga` API (Fig. 2). `xilfpga` did not provide a partial bitstream readback operation, so we modified the read operation to take in both a frame address and number of frames. XBERT also provides a version of write that takes raw frame data and fills in the configuration commands to set frame address and specify a write operation since these are needed to turn raw frame data into a proper partial bitstream. This expansion was necessary to support writes, since we are generating our own frame set and do not have the bitstream configuration command header normally produced by Vivado when it produces a bitstream.

Due to lack of flow control, `xilfpga` readback operations can only run reliably at a lower rate than the 200 MHz peak operation (Sec. VI). The default configuration in the 2019.2 `xilfpga` release sets the clock down to 23.8 MHz for a top throughput of 95 MB/s using the PCAP Clock Generator Configuration (`PCAP_CTRL (CRL_APB)` register). Our experience suggests 150 MHz is likely to work reliably, for a top throughput of 600 MB/s.

VII. Compressing Translation Table

The logical-to-physical translation tables (Sec. III) can become quite large. If we simply stored a 32b frame address and a 16b bit position for each bit in a BRAM, the translation table would be at least 48× larger than the total BRAM data we hope to map. Fortunately, there is some structure. We don't need to store the frame for every bit when multiple bits are in the same frame, as is typical. This can save a factor of 3.

To compress further, we exploit the way the bit positions repeat among frames. To illustrate, let's start by considering the simple case of a single BRAM design. A single frame will hold some number of logical words in its 144 bits. In the case of a 72b-wide logical memory, this could be 2 logical words; in the case of an 8b-wide logical memory, this could be 16 logical words. Define W_{frame} to be this value (2 or 16 above). For each successive group of W_{frame} words, the offset positions within the frame are the same, allowing us to algorithmically determine the frame offset (logical_address/W_{frame}) and reuse a single bit map for the (up to 144) bits in the frame to the offset within the frame. For the case of this single BRAM design, this means we only need to store one frame address (the base) and (up to) 144 bit offset addresses (could do with 12b for the 2976 bits in a frame, but we round to 16b). So, we need at most 32+144×16=2336b instead of 48×36864=1.7 Mbits with no compression.

To be general, we must support the variety of ways that a logical memory can be mapped to physical BRAMs. This can mean multiple BRAMs organized in parallel to support the logical word width (e.g. 8 BRAMs that each supply 4b of a 32b word), multiple BRAMs covering different address ranges to handle deep memories (e.g. 2 BRAMs covering a 36x2048 memory where one handles addresses 0 through 1023 and another handles addresses 1024 through 2047), or combinations thereof. There are even cases where the component BRAMs hold heterogeneous sub-widths and sub-depths (example in Sec. III). So, the general compression case is more involved to deal with these additional irregularities. Nonetheless, there is regularity across frames that can be described algorithmically. This allows us to generalize the single-BRAM observations to form a general compression strategy. Using these observations, we signficantly compress the translation tables (Tab. III).

VIII. Accelerated Translation

As we see in Tab. IV, once we speed up read DMA, translation time is slow compared to DMA transfer times. Even with pre-compiled tables, the code is still extracting and inserting one bit at a time. We implemented a multi-bit accelerated conversion where we pre-compute the impact of a sequence of bits (e.g., a byte) in the word (or frame) to create a vector of bits to be applied to the frame (or words) and store them in a table.

```
for (int b=0;b<bytes_in_word;b++) {
  uint8_t field=(logical>>(b*8))&BMASK;
  physical|=table_lookup[b][field]; }
```

This reduces the processor cycles but demands larger translation tables. Our current implementation supports these translation tables for single-BRAM memories.

IX. Evaluation

As illustrated in Sec. III and highlighted throughout the paper, the size and performance of XBERT transfers are highly design-dependent, API-usage-dependent, and impacted by our optimizations. In this section, we evaluate the performance of the API across several scenarios to concretely characterize the performance implications of these various cases.

A. Methodology

Our experiments use Vivado 2018.3 including SDK. Embedded ARM processor code is compiled `-O3`. Experiments are performed on the Ultra96 v2 board that includes an XCZU3EG

containing 216 RAMB36s. Designs are run on a bare metal configuration. We integrated `xilfpga` source from 2019.2 due to bugs in the 2018.3 version of `xilfpga`.

A primary performance metric is *effective throughput*. Effective throughput accounts for that fact that the useful data is only a subset of the data transferred in frames. For example, if we only want the contents of one RAMB36 in an UltraScale+ device frame, we get 144 relevant bits per frame but must transfer an entire 2976-bit frame.

To illustrate typical application scenarios, we include a couple of complete designs. These give some indication of how frames share BRAMs from multiple logical memories.

- Huffman—our Verilog design from Sec. III-F. It has 4 logical memories consuming 4 RAMB18s and 3,903 LUTs. Each logical memory fits in a single RAMB18.
- Rendering—the Rendering HLS benchmark from the Rosetta Benchmark Suit [35]. It has 11 logical memories consuming 41 RAMB36s, 9 RAMB18s and 11,109 LUTs. Several memories are large, requiring many BRAMs. Some memories have gaps where no bits are defined.

B. Results

1) Raw Bitstream Data Transfers: Tab. II shows raw bitstream transfer times and raw and effective throughputs. This shows the DMA transfer performance and implications without the time required for translation, which is separated out in the next section. Note that accessing all the BRAMs in a frame (single BRAM frame-set rows) provides the highest effective throughput. Since most of the bits in a BRAM frame are BRAM data bits, the effective bandwidth is over half the raw bandwidth. Note that this effective bandwidth is higher than a full bitstream read or write even when we care about every bit of the BRAMs in all the BRAMs on the chip, since the partial read is only reading frames that hold BRAM data.

2) Translation: Tab. III reports the translation table sizes and times and effective throughputs for translating between logical and frame representations.

Tab. III shows that we can reduce translation tables about two orders of magnitudes, into the kilobyte range per BRAM (Sec. VII). Compression slightly slows translation. Acceleration (Sec. VIII) roughly halves translation time, while adding tables that are about as large as the uncompressed tables. Seven of the 11 memories in Rendering are single BRAM memories, but those only account for 4% of the bits, so acceleration makes small impact when translating all the memories.

3) BERT API Operation: Single Memory: Tab. IV summarizes the performance reading and writing a simple 512×64 memory that fills a RAMB36 block. The first line shows the default read speed (Sec. VI) with the second showing the impact of increasing the PCAP DMA read speed to 150 MHz (600MB/s). The third line shows the impact of compression; in this case it slows down translation slightly. The fourth line then shows the impact of acceleration, which roughly halves the translation time.

The next three lines (labeled "Frameset") look at transfering all 12 memories that share the same set of frames. The first case captures the total time when performing separate `bert_read` and `bert_write` operations on each memory; this takes roughly 12 times as long as the single BRAM case since it is just the sum of the individual times, and it achieves the same bandwidth. The middle line uses `bert_transfuse` to either read or write all 12 memories in one operation; as a result, the DMA transfer time is comparable to the transfer time in each of the individual BRAM reads, while the translation time is almost 12 times larger since each memory must be separately translated. Since DMA transfer time was roughly comparable to translation time in the single RAMB36 read or write, this results in roughly double net throughput. The last of the frameset lines shows the impact of performing a single transfuse operation that both reads and writes all the memories.

The final line in the table shows reading and writing a single frame (two 64b words) from the RAMB36. This takes less time than reading or writing the whole memory and translating all the frames. The translation time can be small here. However, since there is considerable fixed time in setting up the DMA operations, the net throughput is low.

4) BERT API Operation: Application Level: The single BRAM and the transfuse of all the BRAMs in a frame set cases in Tab. IV bracket expected typical performance. Tab. V shows the transfuse performance on the applications when we read or write all memories. Huffman shows a 2–3× improvement with transfuse operations. Rendering shows little benefit, with translation taking more time in the transfuse case, possibly due to caching effects for the large frame memory it needs to accommodate all memories. Since the dominant time is typically in translation, total transfusion time is mostly linear in the data being read and written.

X. Conclusions

The configuration path on modern FPGAs provides access to embedded memories. In terms of FPGA resources, it is a lightweight interface to get data in and out of embedded memories. XBERT provides a user-level API that makes using this capability lightweight for the application developer, as well. With XBERT accessing a logical memory is as easy as an API call. This is useful for loading initial memory states at program startup, recovering final data and status at program completion, debugging, and for infrequent data transfers between the FPGA fabric and the embedded cores. The XBERT API takes care of logical-to-physical translations and includes optimizations to compress the necessary translation information and to minimize the data that must be transferred in and out of the FPGA. The XBERT tool flow automates the generation of the translation information needed by the API.

Acknowledgments

Matthew Hofmann was supported by the Vagelos Integrated Program in Energy Research (VIPER) program. Zhiyao Tang was supported by the Rachleff Scholars Program. Jonathan Orgill and Jonathan Nelson were supported by Watson Fellowships. Xilinx donated Vivado tools for use in this work.

TABLE II
RAW BITSTREAM READ AND WRITE TIMES ON XCZU3EG

What	Raw Time (ms)	Raw Thput (MB/s)	1 BRAM bits	Effective Thput (MB/s)	all BRAMs bits	Effective Thput (MB/s)
full bitstream read (24 MHz)	58.50	95.2	36,864	0.079	7,962,624	17.0
full bitstream read (150 MHz)	9.37	594.0	36,864	0.492	7,962,624	106.2
full bitstream write	8.34	667.6	36,864	0.553	7,962,624	119.3
single BRAM frame-set read (24 MHz)	1.01	87.2	36,864	4.560	442,368	54.7
single BRAM frame-set read (150 MHz)	0.20	477.8	36,864	23.000	442,368	276.5
single BRAM frame-set write	0.21	462.2	36,864	22.300	442,368	267.1
single BRAM frame read (24 MHz)	0.10	8.4	144	0.178	1,728	2.1
single BRAM frame read (150 MHz)	0.10	8.4	144	0.178	1,728	2.1
single BRAM frame write	0.10	9.7	144	0.178	1,728	2.1

"frame-set" is the set of 256 frames that contain the contents of a single BRAM (Fig. 3).
24 MHz is the read speed in xilfpga; 150 MHz is the highest speed we were able to reliablity run DMA readback.

TABLE III
TRANSLATION TIME ON XCZU3EG

What	Uncompressed Table Size (B)	Uncompressed Time (ms)	Uncompressed Effective Thput (MB/s)	Compressed Table Size (B)	Compressed Time (ms)	Compressed Effective Thput (MB/s)	Accelerated Table Size (B)	Accelerated Time (ms)	Accelerated Effective Thput (MB/s)
frames→logical: single RAMB18 (x36)	148,832	0.38	6.1	2,128	0.49	4.7	126,920	0.25	9.2
logical→frames: single RAMB18 (x36)	148,832	0.43	5.4	2,128	0.50	4.6	126,920	0.18	12.8
frames→logical: single RAMB36 (x64)	263,520	0.61	6.7	2,352	0.79	5.2	243,808	0.40	10.2
logical→frames: single RAMB36 (x64)	263,520	0.71	5.8	2,352	0.84	4.9	243,808	0.37	11.1
frames→logical: all BRAMs Huffman	313,528	0.96	5.1	4,424	0.98	5.0	320,648	0.61	8.0
logical→frames: all BRAMs Huffman	313,528	1.00	4.9	4,424	1.16	4.2	320,648	0.35	13.9
frames→logical: all BRAMs Rendering	11,847,032	32.84	5.6	91,968	51.71	3.6	413,936	50.70	3.6
logical→frames: all BRAMs Rendering	11,847,032	38.07	4.9	91,968	51.64	3.6	413,936	50.64	3.7

TABLE IV
BERT API SINGLE RAM36 PERFORMANCE ON XZCU3EG

What	Effective Bits (bits)	Clock Freq. (MHz)	Read Time Trans. (ms)	Read Time DMA (ms)	Read Time Total (ms)	Read Effective Thrput (MB/s)	Write Time Trans. (ms)	Write Time DMA (ms)	Write Time Total (ms)	Write Effective Thrput (MB/s)
Uncompressed	32768	24	0.61	1.22	1.84	2.22	0.71	0.25	1.04	3.92
Uncompressed (Sec. VI)	32768	150	0.61	0.42	1.03	3.94	0.71	0.25	1.04	3.91
Compressed (Sec. VII)	32768	150	0.86	0.45	1.32	3.08	0.87	0.27	1.21	3.37
Accelerated (Sec. VIII)	32768	150	0.39	0.45	0.85	4.81	0.28	0.27	0.62	6.59
Frameset, accelerated, separate ops	393216	150	4.67	5.41	10.13	4.85	3.72	3.35	8.10	6.06
Frameset, accelerated, transfuse	393216	150	3.71	0.45	4.18	11.74	3.71	0.46	4.18	11.75
Frameset, accelerated, transfuse	786432	150	Read and Write all →				6.27	0.70	7.00	14.03
Single Frame, accelerated	128	150	0.32	0.30	0.63	0.03	0.00	0.10	0.11	0.14

24 MHz is the read speed in xilfpga; 150 MHz is the highest speed we were able to reliablity run DMA readback.

TABLE V
BERT API APPLICATION PERFORMANCE ON XZCU3EG

What	Effective Bits (bits)	Clock Freq. (MHz)	Read Time Trans. (ms)	Read Time DMA (ms)	Read Time Total (ms)	Read Effective Thrput (MB/s)	Write Time Trans. (ms)	Write Time DMA (ms)	Write Time Total (ms)	Write Effective Thrput (MB/s)
Huffman, accelerated, separate ops	38912	150	0.70	1.67	2.40	2.02	0.38	1.81	2.48	1.95
Huffman, accelerated, transfuse	38192	150	0.45	0.45	0.91	5.30	0.45	0.45	0.90	5.35
Huffman, accelerated, transfuse	77824	150	Read and Write all →				0.86	0.69	1.62	5.99
Rendering, acclerated, separate ops	1480192	150	43.08	7.05	50.19	3.68	45.98	7.41	54.50	3.39
Rendering, accelerated, transfuse	1480192	150	50.70	4.35	55.08	3.35	50.64	4.31	54.97	3.36
Rendering, accelerated transfuse	2960384	150	Read and Write all →				103.08	6.30	110.15	3.35

150 MHz is the highest speed we were able to reliablity run DMA readback.

REFERENCES

[1] K. Heyse, J. Basteleus, B. A. Farisi, D. Stroobandt, O. Kadlcek, and O. Pell, "On the impact of replacing low-speed configuration buses on FPGAs with the chip's internal configuration infrastructure," *ACM Transactions on Reconfigurable Technology and Systems*, Oct. 2015. [Online]. Available: https://doi.org/10.1145/2700835

[2] *XBERT Github Website*, 2021. [Online]. Available: https://github.com/icgrp/bert/

[3] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[4] J. Gray, "GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2016, pp. 17–20.

[5] I. Tili, K. Ovtcharov, and J. G. Steffan, "Reducing the performance gap between soft scalar CPUs and custom hardware with TILT," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 10, no. 3, pp. 22:1–22:23, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3079757

[6] N. Kapre and A. DeHon, "VLIW-SCORE: Beyond C for Sequential Control of SPICE FPGA Acceleration," in *Proceedings of the International Conference on Field-Programmable Technology*. IEEE, December 2011.

[7] P. Yiannacouras, J. G. Steffan, and J. Rose, "Portable, flexible, and scalable soft vector processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 8, pp. 1429–1442, 2012.

[8] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2014, pp. 117–126.

[9] P. Cooke, L. Hao, and G. Stitt, "Finite-state-machine overlay architectures for fast FPGA compilation and application portability," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 3, pp. 54:1–54:25, 2015. [Online]. Available: http://doi.acm.org/10.1145/2700082

[10] V. Sateesh, C. Mckeon, J. Winograd, and A. DeHon, "Pipelined parallel finite automata evaluation," *Proceedings of the International Conference on Field-Programmable Technology*, 2019.

[11] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., and A. DeHon, "GraphStep: A system architecture for sparse-graph algorithms," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006, pp. 143–151.

[12] N. Kapre and A. DeHon, "SPICE^2: Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA," *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, vol. 31, no. 1, pp. 9–22, January 2012.

[13] E. Hung and S. J. Wilton, "Towards simulator-like observability for FPGAs: A virtual overlay network for trace-buffers," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2013, p. 19–28. [Online]. Available: https://doi.org/10.1145/2435264.2435272

[14] J. Goeders and S. J. E. Wilton, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, vol. 36, no. 1, pp. 83–96, 2017.

[15] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2011, pp. 97–106.

[16] *Virtex FPGA Series Configuration and Readback*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, March 2005, XAPP 138. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp138.pdf

[17] S. McMillan and S. A. Guccione, "Partial run-time reconfiguration using JRTR," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, ser. LNCS, no. 1896. Springer-Verlag, 2000, pp. 352–360.

[18] C. Carmichael, M. Caffrey, and A. Salaza, *Correcting Single-Event Upsets Through Virtex Partial Configuration*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2000, XAPP 216. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp216.pdf

[19] M. Happe, A. Traber, and A. Trammel, "Preemptive hardware multitasking in ReconOS," in *Proceedings of the International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ser. LNCS vol. 9040. Springer, 2015.

[20] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong, "Metawire: Using FPGA configuration circuitry to emulate a Network-on-Chip," in *International Conference on Field Programmable Logic and Applications*, 2008, pp. 257–262.

[21] R. le Roux, G. van Schoor, and P. van Vuuren, "Block RAM implementation of a reconfigurable real-time PID controller," in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012, pp. 1383–1390.

[22] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar, "Interdiction in practice—-hardware trojan against a high-security USB flash drive," *Journal of Cryptographic Engineering*, vol. 7, pp. 199–211, 2017. [Online]. Available: https://doi.org/10.1007/s13389-016-0132-7

[23] D. Ziener, J. Pirkl, and J. Teich, "Configuration tampering of BRAM-based AES implementations on FPGAs," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018, pp. 1–7.

[24] K. Dang Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 894–897.

[25] J. Gomez-Cornejo, A. Zuloaga, I. Villalta, J. D. Ser., U. Kretzschmar, and J. Lazaro, "A novel BRAM content accessing and processing method based on FPGA configuration bitstream," *Journal of Microprocessors and Microsystems*, vol. 49, no. C, pp. 64–76, Mar. 2017. [Online]. Available: https://doi.org/10.1016/j.micpro.2017.01.009

[26] "Xilinx Microblaze Soft Processor Core," Webpage, 2012, http://www.xilinx.com/tools/microblaze.htm.

[27] *Vivado Design Suite User Guide, Embedded Processor Hardware Design*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, December 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug898-vivado-embedded-design.pdf

[28] *Configuration Readback Capture in UltraScale FPGAs*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, November 2015, XAPP 1230. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1230-configuration-readback-capture.pdf

[29] "Project X-Ray: Documenting the Xilinx 7-series bistream format," https://github.com/SymbiFlow/prjxray, 2020.

[30] "Project U-Ray: Xilinx UltraScale bitstream documentation," https://github.com/SymbiFlow/prjuray-db, 2020.

[31] B. Nelson and J. Orgill, "Project X-Ray BRAM patch," https://github.com/SymbiFlow/prjxray-bram-patch, 2020.

[32] *Vitis Unified Software Development Platform 2020.2 Documentation: Managing Interface Synthesis*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, March 2021. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/managing_interface_synthesis.html

[33] *UG909: Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, December 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug909-vivado-partial-reconfiguration.pdf

[34] "Xilfpga," https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841910/Xilfpga, 2020.

[35] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.