BERT: Bitstream Embedded RAM Transfusion

Omitted for Blind Review

***Abstract*—Most current FPGA devices include a large number of embedded SRAM hard blocks that can provide high memory bandwidth to the application. In many applications, a central controller needs occassional, but not continuous, access to read or write these memories. Common scenarios include loading programs into instruction stores for programmable overlays, snapshotting memory state for debugging, configuring debug overlays, loading test data and offloading test results, and saving and restoring memory state. Providing access to these memories from a central controller consumes memory ports, bandwidth, and expensive configurable routing resources. Fortunately, the FPGA device has already committed hardwired resources to allow read/write access to these memories through the bitstream reconfiguration path that is independent of the memory ports and configurable routing available to the user-level application. Unfortunately, this path operates at a low level that requires deep understanding of how the logical memories are mapped onto the physical SRAM blocks. In this paper we present BERT, an API and design toolset for zero-cost access to the on-chip SRAM blocks using the device's configuration path. The BERT API is high-level, allowing developers to specify data transfers in terms of logical memories in the user-level application. BERT is efficient, consuming zero reconfigurable resources and with no impact on Fmax. BERT achieves a transfusion bandwidth of megabytes per second and complete readback of an isolated 36Kb block RAM in less than 6 ms on a Xilinx UltraScale+ MPSoC Zynq.**

I. Introduction

Today's FPGA fabrics include a large number (hundreds to thousands) of distributed, embedded RAM blocks. These RAMs can operate in parallel, providing high aggregate memory bandwidth and low latency to adjacent computational 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 or to be stored in other memory in the computing system.

We can 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 and limited high-speed channels. It also consumes the limited read/write ports on the embedded RAM primitives. This extra logic can slow down the memories or create additional challenges to meet application timing requirements.

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, the format of data used for bitstream programming is not the same as the logical format for the stored data. Even 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.

To address these needs, we provide a Bistream Embedded RAM Transfustion (BERT) API to provide a high-level, logical interface to read and write the contents of logical memories mapped on the FPGA fabric using partial reconfiguration (Sec. IV). Our automatic tool flow extracts the information about how logical memories are mapped to physical embedded RAMs and embeds code to translate the data to support the BERT API. Our API is able to read and translate data in a few milliseconds providing effective data tranfer rates in the MB/s. This is not as fast as a dedicated, high-speed link (e.g. an AXI channel that might run over 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 complete) or at modest bandwidth (e.g., periodical parameter adjustments) as illustrated in Sec. III. 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 API for the UltraScale+ MPSoC Zynq including both the tool flow to extract the logical↔physical mapping for individual memory bits and runtime support (Sec. VI)
- A characterization of the performance of the API implementation (Sec. VIII)

II. Background

As early as 2000 Xilinx provided experimental, low-level support for partial reconfiguration readback and reloading of embedded RAMs [1], [2]. In this time frame it was used for memory readback and scrubbing [3].

Recent work advocates this approach of using bitstream readback and edits to read and write BRAM contents and demonstrates their use to copy data between BRAMs [4]. This work suggests how to reverse engineer the mapping between BRAM bits and their position in the bitstream, but does not provide a complete description or high-level tool that will allow a developer to map their HLS or RTL memory contents into the bitstream or extract bitstream contents and reconstruct the state of the HLS or RTL memory. This highlights the need

for design-level tools to support bitstream embedded RAM transfusion.

ReconOS uses bitstream reads and writes for multiasking [5], but does not provide an interface to convert to or from logical contents.

Intel provides an In-System Memory Content Editor that allows readback and editing of embedded RAM content, but it consumes a port to memories and uses JTAG rather than using the configuration path [6].

Under linux on Zynq devices, it is possible to map a memory device through `/dev/mem` [7]. However, this requires that the design be prepared with an AXI channel dedicated to provide access to the BRAM.

A. Existing Bitstream Manipulation Support

Xilinx Vivado provides low-level support for changing the initial value of configuration memories in a bitstream mostly to support instruction memories for MicroBlaze processors [8]. 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 [9]. The MMI generated by Vivado does not cover all logical memories in the design, and `UpdateMEM` only produces a complete bitstream.

Xilinx tools also produce a Logic Location (LL) file for memories with initial values as part of bitstream readback generation [10]. 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 they are mapped onto multiple physical BRAM tiles, or the role of each of the bits in the physical BRAM (e.g. which is bit 0 of a word, which corresponds to the 23rd location in the memory). Nonetheless, the LL file is useful for deciphering the frame and bit positions for a BRAM.

B. Project X-Ray

Project X-Ray [11] provides a database mapping the configuration bits in Xilinx 7-series devices, including the frame and bit locations for memories. They also provide tools for converting between raw bitstreams and frames and a logical view of the bitstream data in an FPGA Assembly (FASM) form. This contains no information on logical mapping of an application's memory to physical BRAMs, but does provide information and tools for bitstream encoding. Project X-Ray does not cover Xilinx UltraScale+ devices.[1]

C. The BYU bram-patch Tool

BYU previously developed an open-source tool, `prjxray-bram-patch`, [12] based on the Project X-Ray database and associated tools that serve a similar role as Vivado's `UpdateMEM`, but works for all logical memories in a design, not just the ones in MicroBlaze or using XPM macros. The `bram-patch` tool defines a Memory Description Data (MDD) file that plays a similar role to the Xilinx MMI file. It describes, for each logical memory in a design, the collection of physical BRAMs to which the logical memory is mapped. They have developed a tool, `ExtractMDD`, that uses the Vivado TCL interface to extract the MDD file information from any design in Vivado. They further have created another tool, `bitMapping` which computes, for each bit in an RTL logical memory, the BRAM tile and INIT string location of that bit, as well as the frame and bit position within the bitstream of that bit. Like `UpdateMem`, `prjxray-bram-patch` only produces a complete bitstream and is designed to run on a separate workstation before the application begins running on the FPGA.

D. Embedded SoC FPGAs

System-on-a-Chip (SoC) components that include processors and FPGA logic are now common, including Intel Arria, MicroSemi PolarFire, and Xilinx Zynq. Building on the bitstream mapping information from Xilinx (Sec. II-A), we focus specifically on the Xilinx UltraScale+ Zynq series SoC FPGAs. In addition to the FPGA fabric, UltraScale+ Zynq SoCs have multiple ARM cores (four A53 cores plus two R5 cores in the UltraScale+ series) and AXI channels between the processing cores and the FPGA fabric. Applications commonly split tasks between the ARM cores and accelerators and interfaces on the FPGA fabric. On the UltraScale+ parts, a single AXI channel can move data in and out of the FPGA fabric at speeds up to 5.3 GB/s per direction [13].

III. Customers of On-Line Transfusion

As the MicroBlaze case illustrates, programming BRAM-configured overlay architectures are one obvious need to modify embedded RAM contents. For embedded soft processors with small instruction memories, a simple BRAM memory can be adequate to hold the entire program, avoiding the expense of linking the BRAM memory into a cache hierarchy. These embedded soft processors may include RISC-V processor cores [14], [15], custom VLIWs [16], and customized Vector [17], [18] processors. More generally, this need to load instructions also applies to loading configurations for more specialized overlay fabrics such as dedicated FSM evaluators [19] and Neural Networks [20]. In Software-Defined Networking, including OpenFlow and P4 switches [21], [22], most packet processing and switching is based on a series of table lookups. The main dataplane computation is based on read-only access to these tables. Tables are updated occassionally from the control plane, usually by a control processor.

For at-speed unit testing of FPGA building blocks, we need to feed the module-under-test with data at full rate and capture the results. This can easily be done with a memory to source data into the module-under-test and a second memory to record the results. The speed at which we load these source and result memories is not important. Similarly, in live debugging when

[1] As this paper was being finalized a Project U-Ray was released which describes itself as "Documenting the Xilinx Ultrascale, Ultrascale+ and UltraScale MPSoC series bit-stream format." An alternate backend could use Project U-Ray in place of the LL files used in our initial implementation.

we include an Internal Logic Analyzer (ILA) or trace buffers [23], [24] to capture data during operation, the speed at which we offload this data after a test is often not critical.

In multi-context environments, it may be necessary to save and restore the memory state of an FPGA computation [5], [25], [26]. While this can be done at a low-level by saving and restoring all the BRAM data without regards to what is live and what is changing, it may often be possible to reduce the cost of saves and restores by saving only the logical memories that hold volatile data and only restoring the logical memories needed for the task. To move a computation between implementations (e.g., between hardware and emulation as in [27]), it is also necessary to extract the logical state.

At the extreme, the BERT functionality could provide an inexpensive way to support advanced abstractions like CoRAM [28] 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 BERT, the same functionaliy can be provided using the existing reconfiguration path hardware support without adding overlay logic.

IV. 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 devices, each frame has 93 32b words. Since BRAM data constitutes a small fraction of the bits in the total bitstream, using partial reconfiguration to access only BRAM frames reduces bitstream read or write time compared to full bitstream reads or writes. Furthermore, accessing only the frames associated with the data that needs to be read or written further decreases time and increases bandwidth.

A. BRAM Configuration Frames

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 [29, 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. 1). The block 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.

B. PCAP

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

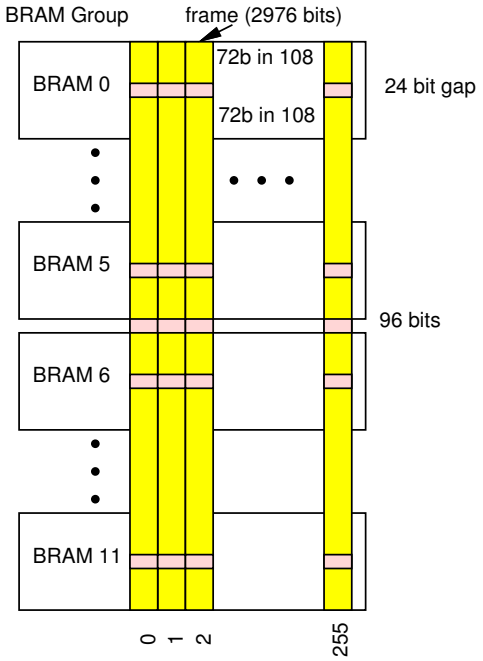

Fig. 1. Frame Organization for UltraScale+ BRAM Contents

with a peak operation of 200 MHz supporting up to 800 MB/s [29, Chapter 8, Configuration Time]. However, that peak speed does not appear sustainable on readback as noted below.

C. xilfpga

Xilinx provides the `xilfpga` library [30] for performing DMA bitstream transfers on Zynq UltraScale+ components. It allows full bitstream load and readback and partial bitstream load for `.bit` files that include a header of control commands to the PCAP. It does not provide partial readback or partial bitstream writes for raw frame data that lack the command header.

Due to lack of flow control, readback operations can only run reliably at a lower rate. 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. Our experience suggests 150 MHz is likely to work reliably, for a top throughput of 600 MB/s. Above 150 MHz, bits are lost and readback DMA operations will time out failing to receive the expected data set.

V. API

BERT is designed to run on the embedded APU cores on Zynq SoCs. The entry-level BERT interface is a simple pair of routines to either read out an entire logical memory into an array (`bert_read`) or to update a memory with the contents of a provided array (`bert_write`).

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

These 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. This simple interface can only handle arrays up to 64b wide, limited by C scalar datapaths. Wider memories will need to use the more full featured API described below.

To support the API, BERT tools produce files `mydesign.h` and `mydesign.c` that include definitions of the specific memories in this design that can be linked into the user application as part of the board-support package. The header (`mydesign.h`) defines the names of the logical memories in the FPGA design as macros (`#defines`) for memory indicies. To access a memory, provide the macro for the desired logical memory for the `logicalm` argument in the BERT API function.

Another limitation of the simple API above is that it combines translation with bitstream read and write operations. This can be inefficient if the application needs to read or update many, different logical memories that are mapped to the same frames. In this case, the simple API may read the frames redundantly or update different portions of the same frames in successive calls. This can be inefficient in runtime.

To allow more efficient transfer, BERT also supports an API that can read and write a set of multiple memories. This is provided by the `bert_transfuse()` call, which allows BERT 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);
```

`bert_transfuse` takes an array describing the operations on a collection of logical memories. To describe each transfusion operation, it uses a structure:

```
struct bert_meminfo {
   int logical_mem;
   int operation;
   uint64_t *data;
   int start_addr;
   int data_length;
};
```

Here, we specify the memory (`logical_mem`), the operation (read or write), where to get (or store) the logical memory data (`data`), 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. The compiled, design-specific header data will capture the actual width of data words for each logical memory. Here, 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 assumes that it is safe to perform the read and write operation. Putting the logic or computation into a safe state where the memories are not being written during the read or write is the responsibility of the application. The BERT API does not know anything about the design to provide that service.

VI. Design Flow

A. Mapping to Physical BRAMs

Given a specific design, we first determine (1) how each of the logical memories in that design is mapped to BRAM primitives in the physical device and (2) where in those physical BRAMs each bit of the logical memory can be found. We use the `mdd_make.tcl` script from the `prjxray-bram-patch` tool [12] to extract a `.mdd` file, from which the above mapping information can be computed. The `.mdd` file identifies the BRAM primitive(s) used for each logical memory. For each BRAM primitive the `.mdd` file also specifies (1) the range of words and bits from the original logical memory that are contained within that BRAM, (2) the read and write widths for the BRAM, (3) how the data bits are spread across the regular and parity bits of the BRAM, and (4) and whether the primitive is a RAMB18 or RAMB36.

B. Building a Device Database

To provide a general solution working entirely from Xilinx provided infomation, we create a database of BRAM frame and bit locations.[2] For a particular design, the Xilinx Logic Location (LL) files will tell us the frame and offset within the frame for the bits in a particular used physical BRAM location. We produce a design with a single 72b×512 location memory that maps to a single RAMB36 (or a 36×512 to map to a single RAMB18), then generate a series of placement constraints that place the memory in each of the BRAM locations on a particular Xilinx device. Collecting the resulting LL files, we can build a database of the frame and bit locations for every BRAM in the the device.

C. Tool Flow Building Applications

To prepare designs for BERT, we compile a per-design header file `mydesign.h` and memory data file `mydesign.c` that contains the physical-to-logical translation table. This combines the result of the physical bit mapping (Sec. VI-A) and the frame and location data extracted from Xilinx Logic Location files (Sect. VI-B) into a logical bit to physical frame and offset location translation table (See. Fig. 2).

[2]Where available, databases like Project X-Ray and U-Ray could be used instead.

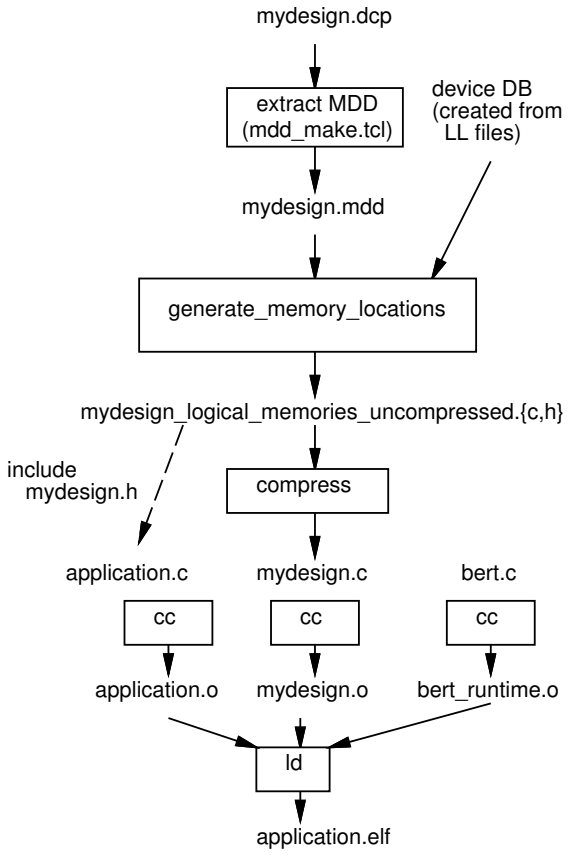

Fig. 2. Tool Flow to produce per-design header file and translation table with logical-to-physical translation data

D. Runtime Operation

During operation, the BERT API operations behave as follows. `bert_read` first looks up the range of frames for the BRAM, then performs a DMA partial bitstream read using our extended version of `xilfpga` described in the next section, and finally performs a physical-to-logical translation to extract the data into an array. `bert_write` performs a logical-to-physical mapping to setup BRAM frame contents. It also resets the write-enable bits for the RAMB36 memories associated with the logical memory in the BRAM frame that will be written [5], leaving the write-enable bits unset for the RAMB36s in the frame that are not changing. Finally, it then performs a partial reconfiguration DMA write to write the modified frame contents back to the FPGA fabric. In the case of partial frame write, including BRAM18 writes, `bert_write` reads the existing frame contents first.

E. Expanded xilfpga API

To support these operations, we had to provide expanded versions of routines in the `xilfpga` API. `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:

```
u32 XFpga_GetPlConfigData(
    XFpga *InstancePtr,
    UINTPTR ReadbackAddr,
    u32 NumFrames, u32 FrameAddr)
```

We provided a version of write that takes raw frame data and fills in the configuration commands to set frame address and specify a read operation since these are needed to turn raw frame data into a proper partial bitstream.

```
u32 XFpga_PL_Frames_Load(
  XFpga *InstancePtr,
  UINTPTR ReadbackAddr, u32 Flags,
  u32 WrdCnt)
```

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.

F. Compressing Translation Table

The translation tables (Sec. VI-C) can become quite large. If we simple 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 up to a factor of 3. While the logical to physical bit mapping can vary widely based on the memory organization and how logical memories are spread across BRAMs, once the word address becomes large enough to access the second frame for all the BRAMS in a word, the bit positions will repeat and the frame addresses are predictable based on offset, at least until the end of a physical set of BRAMs used along the width of a logical memory. Using these observations, we signficantly compress the translation tables–typically by two to three orders of magnitude (Tab. II).

G. Accelerated Translation

As we see in Tab. III, 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 prototyped a table-based accelerated conversion where we pre-compute the impact of a sequence of bits in the word (or frame) to create a vector of bits to be applied to the frame (or words). This allows us to translate multiple bits per lookup, but demands larger translation tables. It appears viable for small (e.g., single BRAM) memories where the translations are relatively compact and have high regularity. It may not be a net win for large memories that span multiple BRAMs. We add accelerated read and write operations to the `operation` types in the `bert_meminfo` structure used by the `bert_transfuse` API call (Sec. V) as well as additional parameters for the accelerated translation tables.

VII. Use Demonstration Examples

As a simple example to demonstrate the uses of BERT, consider developing a Huffman encoding accelerator. The Huffman encoder will take in a stream of bytes and produce a

compressed bitstream mapping each byte to a variable-length bit sequence. For good compression, the Huffman code should be tuned to the data being encoded. We design the encoder to work for any encoding by using an encoding table that provides the bit length and encoding for each uncompressed byte. We can store this encoding table in a BRAM and program for different encodings by loading the memory contents of this encoding table BRAM.

Our first need is to load the encoding table before we begin encoding an input stream. The encoding table is read-only during normal operation. So, we can use `bert_write` to load the encoding table. This takes 1 ms to load the table. Without BERT, we would need to use an AXI port to perform the load, adding 830 LUTs and 640 registers to the design in additional to consuming AXI port bandwidth. Alternately, we could have used Xilinx's `UpdateMem` to update the memory contents. `UpdateMem` takes approximately 4 seconds to produce a new full bitstream that then takes 9 ms to load onto the device. And, `UpdateMem` must be run on a separate host machine that can run the Vivado tools.

When we first design the encoder, we often want to test it for functionality and speed before the data source and data consumer are developed or optimized. We can do that by setting up a memory to hold the input data and a memory to store the output data. Using BERT we need to add nothing else in order to test the design. We can load the input memory with `bert_write` and recover the output with `bert_read`.

To see if we are using the correct encoding for our data, we could also compute a histogram of the input bytes. This can be implemented with a simple 256-element memory where each element holds a count for one of the byte values. We can read the contents of this memory back in 1 ms using `bert_read`. If we're processing one byte per cycle, this histogram will need to perform both a read and a write on each cycle, meaning both ports of the BRAM are in use for the application. For a non-BERT design to provide readback, it would need to share one of the ports or duplicate the memory to effectively provide another read port. This kind of histogram can also be used to capture events for performance monitoring on designs and similarly queried without requiring any FPGA fabric resources.

If we suspect our histogram is not performing correctly (perhaps our logic to deal with the case of back-to-back updates on the same byte isn't quite right), we could "single-step" the histogram by providing one byte at a time to the encoder and using a `bert_read` to capture the histogram memory after each byte is applied. While a histogram is simple, this trick is useful when dealing with complicated logic that maintains internal state like an LZW-encoding tree.

Ultimately, we may want to adaptively update our encoder. We could take the values read back from the histogram and use them to compute a new, optimal Huffman encoding. Then, we can use `bert_write` to apply the new code. Since we use the BERT API, this requires no more fabric or AXI resources for I/O than the static case where the code never changes. Computing the Huffman code takes 1 ms, comparable to the 2 ms to read the histogram and write the new code taken by the BERT API. Since these read and write operations are in the critical path for the program, we used the accelerated translation to bring the the new code write from 0.89 ms to 0.33 ms, such that BERT read and write operations are comparable to the Huffman code computation.

VIII. Evaluation

A. Methodology

All our experiments are performed using Vivado 2018.3 including SDK. 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.

B. Results

1) Raw Bitstream Data Transfers: Tab. I shows raw bitstream transfer times and throughputs. 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 but must transfer an entire 2976 bit frame. Note that accessing all the BRAMs in a frame (single BRAM frame-set read lines under "all BRAMs") 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. Furthermore, 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. II reports the times and throughputs for translating between logical and frame representations.

To illustrate typical application scenarios, we include a couple of complete designs. These give some indication of how component logical BRAMs are arranged in the same or different frames and how frames share BRAMs from multiple logical memories.

- Huffman – this is our Verilog design from Sec. VII. It has 4 logical memories consuming 3 RAMB18s, one RAMB36 and 4,989 LUTs.
- Rendering – this is the Rendering HLS benchmark from the Rosetta Benchmark Suit [31]. We include 9 logical memories consuming 32 RAMB36s, 7 RAMB18s and 11,109 LUTs. The design includes 2 more logical memories that we omit because their mapping is currently more complex than our tools can handle.

Tab. II shows that we achieve two orders of magnitude space compression both on simple designs that map to a single BRAM and to the sample applications. It also shows that the compressed encodings run about 20% faster than the uncompressed designs. With the compressed data structures in the KB range, while uncompressed designs are in the MB range, the compressed data structures will fit in the L1 cache (32KB), while the uncompressed designs may not even fit in

TABLE I
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 (default)	58.50	95.2	36,864	0.079	7,962,624	17.0
full bitstream read (accelerated)	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 (default)	1.01	87.2	36,864	4.560	442,368	54.7
single BRAM frame-set read (accelerated)	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 (default)	0.10	8.4	144	0.178	1,728	2.1
single BRAM frame read (accelerated)	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 256 frames that contain the contents of a single BRAM
(default) is the read speed in xilfpga; (accelerated) is the highest speed we were able to reliablity run DMA readback

TABLE II
TRANSLATION TIME ON XCZU3EG

What	Uncompressed			Compressed		
	Table Size (B)	Time (ms)	Thput (MB/s)	Table Size (B)	Time (ms)	Thput (MB/s)
frames→logical: single RAMB36	290,000	5.27	0.87	2,000	4.26	1.08
logical→frames: single RAMB36	290,000	5.44	0.85	2,000	4.24	1.09
frames→logical: all BRAMs HuffmanCycle	310,000	5.04	0.84	2,800	4.19	1.01
logical→frames: all BRAMs HuffmanCycle	310,000	4.98	0.85	2,800	4.08	1.04
frames→logical: all BRAMs Rendering	8,880,592	164.04	0.84	97,744	132.19	1.05
logical→frames: all BRAMs Rendering	8,880,592	163.70	0.84	97,744	132.32	1.04

the L2 cache (1 MB). Even with compression, the translations are slower than the data transfer and are the bottleneck in the memory insertion and extraction process. None of the data bit locations are contiguous in the frame layout, meaning every bit must be independently deposited into or extracted from the frame word.

3) BERT API Operation: Tab. III shows the composite performance achieved including translation and bitstream transfer for specific cases and interfaces.

Single frame or single word reads and writes can be expensive with `bert_read` and `bert_write` since they process the entire logical memory. Here, `bert_transfuse` can be more efficient by reading only the single frame of interest.

Reading everything in a frameset is also more efficient with `bert_transfuse` because it can read the frameset once, whereas `bert_read`/`bert_write` will need to read the entire frameset for each memory.

Without the write enable, it is necessary to perform a read of a frame before the write in order to preserve the non-changing BRAMs in the frame. Comparing the write enable (we) cases and the no write-enable cases (no we) cases, we can see the benefits that come from the architectural inclusion of the write-enable bits.

Transfusion also benefits contiguous multi-frame write sets by allowing the writes to occur as a single DMA operation rather than starting multiple DMAs. This is best seen in the Huffman and Rendering examples comparing the write-enable `bert_writes` to the `bert_transfuse` writes with write enables.

We achieve our highest read bandwidth with dense transfuse reads and our highest write bandwidth with dense transfuse writes exploiting write enables as shown in the "all BRAM in frameset" cases. The designs come 50–90% close to this representing typical packing density of BRAMs into frames.

IX. DISCUSSION AND FUTURE WORK

a) Use: At the current, translation-limited speeds, this is most useful for debugging, statistics collection, and startup configuration.

b) Status: The current implementation doesn't fully support RAMB18s and a few corner cases for BRAM mappings including cases where logical memory bits are replicated.

c) Acceleration: Translation is the bottleneck. Further tuning of accelerated translation (Sec. VI-G) may offer additional gains. For example, it may be possible to use NEON instructions to operate on 128b vectors to further reduce translation time. The translation process is independent for frames, meaning an opportunity to exploit data-level parallelism on the processor cores. The UltraScale+ Zynq has 4 cores. With the compressed encoding, we are not memory bound to main memory, and we can jump forward to any of the repeats, making it amenable to distribute translation to cores by address and frame range.

TABLE III
BERT API PERFORMANCE ON XZCU3EG

What	Time			Effective	
	Trans. (ms)	DMA (ms)	Total (ms)	bits	Thrput (MB/s)
single BRAM frame transfuse read	0.02	0.42	0.44	144	0.04
single BRAM frame transfuse write (preserve others, no we)	0.02	0.41	0.43	144	0.04
single BRAM frame transfuse write (preserve others, we)	0.02	0.11	0.44	144	0.04
single BRAM read	4.30	1.32	5.61	36,864	0.82
single BRAM write (preserving others, no we)	4.36	1.55	5.93	36,864	0.78
single BRAM write (preserving others, we)	4.36	0.23	4.64	36,864	0.99
single 512x64 write (preserving others, we)	3.88	0.23	4.29	32,768	0.95
single 512x64 write (accelerated)	1.20	0.23	1.61	32,768	2.54
all BRAM in frameset read	51.12	15.84	67.15	442,368	0.82
all BRAM in frameset transfuse read	50.70	1.55	52.40	442,368	1.06
all BRAM in frameset single writes (no we)	52.32	18.60	71.72	442,368	0.77
all BRAM in frameset single writes (with we)	52.32	2.76	56.12	442,368	0.99
all BRAM in frameset tranfuse (no we)	51.17	1.55	52.87	442,368	1.05
all BRAM in frameset tranfuse (with we)	51.06	0.23	51.45	442,368	1.07
Huffman encode write for HuffmanCycle (transfuse we)	0.65	0.13	0.89	5,120	0.72
Huffman encode write for HuffmanCycle (accelerated)	0.11	0.13	0.33	5,120	1.94
all BRAMs in HuffmanCycle bert_read	4.22	3.68	8.12	33,792	0.52
all BRAMs in HuffmanCycle transfuse (read)	4.19	3.37	7.56	33,792	0.56
all BRAMs in HuffmanCycle bert_write (no we)	4.20	4.40	9.03	33,792	0.47
all BRAMs in HuffmanCycle bert_write (we)	4.20	0.72	5.42	33,792	0.78
all BRAMs in HuffmanCycle bert_transfuse (write no we)	4.08	3.88	8.39	33,792	0.50
all BRAMs in HuffmanCycle bert_transfuse (write we)	4.08	0.62	5.17	33,792	0.82
all BRAMs in Rendering bert_read	132.19	13.48	145.58	1,105,920	0.95
all BRAMs in Rendering transfuse (read)	133.81	6.20	140.01	1,105,920	0.99
all BRAMs in Rendering bert_write (no we)	132.32	15.83	149.58	1,105,920	0.92
all BRAMs in Rendering bert_write (we)	132.32	2.35	136.25	1,105,920	1.01
all BRAMs in Rendering bert_transfuse (write no we)	132.22	6.97	139.93	1,105,920	0.99
all BRAMs in Rendering bert_transfuse (write we)	132.22	0.84	133.79	1,105,920	1.03

Reads here are performed at default xilfpga read speeds.

Certain, restricted BRAM contents could be translated more quickly, including cases where the logical memory is filled with a single value. Transfer and translation could be split, minimizing the time for the read or write and allowing the translation to run in parallel with other operations, where the application permits.

d) Generality: The methodology of using LL files to map BRAMs should apply to any Xilinx family. Alternately, it is possible to use the Project X-Ray database for 7-series devices and Project U-Ray for UltraScale+ devices as they become available.

It should be possible to make the BERT API work with other devices with suitable backend drivers. To work with SoCs, it is necessary to get the PCAP DMA transfers working. The Xilinx code for PCAP bitstream loading on 7-series Zynqs is currently listed as deprecated [32]. To work with PCIe-based devices, it will need an interface to the MCAP interface. An interesting extension will be to port the BERT API support to the Xilinx Runtime (XRT) stack for use with both embedded and data center devices, where the data center devices expose the partial reconfiguration capabilities of the device.

e) Open-Source Release: We plan to release our tool flow and BERT API as an open-source package on github and include the URL in the camera-ready version of the paper.

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. BERT provides a user-level API that makes using this capability lightweight for the application developer, as well. With BERT, 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 BERT 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 BERT tool flow automates the generation of the translation information needed by the API.

REFERENCES

[1] *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

[2] S. McMillan and S. A. Guccione, "Partial run-time reconfiguration using JRTR," in *FPL*, ser. LNCS, no. 1896. Springer-Verlag, 2000, pp. 352–360.

[3] 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

[4] 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," *J. Microproc. and Microsys.*, vol. 49, no. C, pp. 64–76, Mar. 2017. [Online]. Available: https://doi.org/10.1016/j.micpro.2017.01.009

[5] M. Happe, A. Traber, and A. Trammel, "Preemptive hardware multitasking in ReconOS," in *Proc. Intl. Conf. on Applied Reconf. Comp.*, ser. LNCS vol. 9040. Springer, 2015.

[6] *Intel Quartus Prime Standard Edition Handbook Volume 3*, Intel Corporation, November 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts-qps-5v3.pdf

[7] "Accessing BRAM in linux," https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842412/Accessing+BRAM+In+Linux, 2020.

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

[9] *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

[10] *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

[11] "Project x-ray: Documenting the xilinx 7-series bistream format," https://github.com/SymbiFlow/prjxray, 2020.

[12] B. Nelson and J. Orgill, "Project x-ray bram patch," https://github.com/SymbiFlow/prjxray-bram-patch, 2020.

[13] *UG1085: Zynq UltraScale+ Device: Technical Reference Manual*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, August 2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

[14] 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.

[15] J. Gray, "GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *FCCM*, 2016, pp. 17–20.

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

[17] P. Yiannacouras, J. G. Steffan, and J. Rose, "Portable, flexible, and scalable soft vector processors," vol. 20, no. 8, pp. 1429–1442, 2012.

[18] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *FPGA*, 2014, pp. 117–126.

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

[20] 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 *FCCM*. IEEE, 2006, pp. 143–151.

[21] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proc. ACM/IEEE Symp. ANCS*, 2008, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477944

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[23] E. Hung and S. J. Wilton, "Towards simulator-like observability for FPGAs: A virtual overlay network for trace-buffers," in *FPGA*, 2013, p. 1928. [Online]. Available: https://doi.org/10.1145/2435264.2435272

[24] J. Goeders and S. J. E. Wilton, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Trans. Computer-Aided Design*, vol. 36, no. 1, pp. 83–96, 2017.

[25] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *FPL*, Aug 2005, pp. 223–228.

[26] K. Rupnow, W. Fu, and K. Compton, "Block, drop or roll(back): Alternative preemption methods for RH multi-tasking," in *FCCM*, April 2009, pp. 63–70.

[27] S. Attia and V. Betz, "StateMover: Combining simulation and hardware execution for efficient FPGA debugging," in *FPGA*, 2020, pp. 175–185. [Online]. Available: https://doi.org/10.1145/3373087.3375307

[28] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *FPGA*, 2011, pp. 97–106.

[29] *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

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

[31] 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 *FPGA*, 2018, pp. 269–278.

[32] Xilinx, "Solution Zynq PL programming," February 2020. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841750/Solution+Zynq+PL+Programming