AxBA: An Approximate Bus Architecture Framework

Jacob R. Stevens, Ashish Ranjan, and Anand Raghunathan
School of Electrical and Computer Engineering, Purdue University
{steven69,aranjan,raghunathan}@purdue.edu

ABSTRACT

Modern computing platforms expend significant amounts of time and energy in transmitting data across on-chip and off-chip interconnects. This challenge is exacerbated in prevalent data-intensive workloads such as machine learning, data analytics and search. However, these workloads also present a unique opportunity in the form of intrinsic resilience to approximations in computations and data. We explore approximate compression of communication traffic, which leverages this intrinsic resilience to improve communication bandwidth and reduce the energy consumed by interconnects. Specifically, we propose AxBA, an approximate bus architecture framework that is aware of the data amenable to approximations and seamlessly compresses/decompresses the corresponding transactions on the bus without requiring any changes to pre-designed masters and slaves. AxBA uses a lightweight compression scheme based on approximate deduplication, which is suitable for the tight latency constraints imposed by bus-based interconnects. To facilitate software development on AxBA-based systems, we introduce a software interface that enables programmers to identify regions of the system address space that are amenable to approximations. We also propose a runtime quality monitoring framework that automatically determines the error constraints for the identified regions such that a specified application-level quality is maintained. We demonstrate the feasibility of the proposed concepts by realizing a prototype AxBA system on a Cyclone-IV FPGA development board using an Intel Nios II processor-based SoC. Across a suite of six machine learning benchmarks, AxBA obtains an average improvement in system performance of `29%` and a `25%` reduction in system-level energy for a `0.5%` loss in application-level quality.

1 INTRODUCTION

The exponential growth in creation and consumption of various forms of digital data has led to the emergence of new application workloads such as machine learning, data analytics and search. These workloads process large amounts of data and hence pose increased demands on the on-chip and off-chip interconnects of modern computing systems. Therefore, techniques that can improve the energy-efficiency and performance of interconnects are becoming increasingly important.

This work was supported in part by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and in part by the National Science Foundation under grant no. 1423290

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '18, November 5–8, 2018, San Diego, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240782

Several circuit and architectural techniques have been proposed to realize faster or more energy-efficient interconnects. Complementary to these approaches, which attempt to reduce the time or energy taken to transfer each bit, communication traffic compression attempts to reduce the volume of data that must be transferred. It leverages the redundancy in communication traffic, *e.g.*, the similarity of successive values transferred across the communication network, to optimize both energy and performance. Prior efforts have explored lossless compression in the memory hierarchy [1, 16, 21] or the interconnects [3, 5, 6, 13, 25] for both general-purpose and embedded computing systems.

Many prevalent workloads exhibit significant *intrinsic resilience*, which is the ability to produce outputs of acceptable quality despite approximations to the underlying computations or data. Approximate computing leverages this property to improve the energy and performance of computing systems at different levels of abstraction, including software, architecture and circuits [23]. This approach opens up a unique opportunity to address the communication bottleneck by adopting *approximate communication traffic compression* wherein the data transmitted on the interconnects is compressed by introducing approximations, yielding additional benefits in energy and performance. We propose AxBA, a framework to realize this concept in bus-based communication architectures, and demonstrate its hardware feasibility through an FPGA prototype system.

Most previous efforts in approximate computing have focused on approximating computations (*e.g.*, by lowering precision or skipping selected computations) or approximating data stored in memory (*e.g.*, lowering refresh rates), while maintaining full accuracy for the communication traffic [4, 9, 14, 18, 19, 22, 24]. A few recent efforts on approximate communication can be classified into two distinct categories: *approximate memory compression* and *approximate interconnects*. Approximate memory compression utilizes techniques to compress data at different levels of the memory hierarchy (in the processor's load/store queues [8], the on-chip cache hierarchy [20], or the off-chip memory controller [17]). These techniques are solely limited to the memory traffic and do not perform any approximations to other data transfers, *e.g.*, transfer of data to/from accelerators, sensors and other peripheral components. Moreover, these proposals require significant changes to existing IPs such as processor cores, caches, and memory controllers, which may be infeasible for hard IP cores, or incur significant (re-)design effort. In contrast, our proposal is transparent to the underlying hardware components in the system and applies to all communication traffic.

In the context of approximate interconnects, previous efforts [7, 10, 15] have proposed approximations to off-chip serial buses that reduce the switching activity on the bus, thereby reducing power consumption. These techniques solely focus on reducing the switching energy and do not improve the volume of bus traffic, which is the focus of our work. Finally, some recent efforts [2, 12] propose to reduce the data traffic in Network-on-Chips (NoCs) by compressing

similar values in a packet or dropping congested packets altogether. These efforts demonstrate the potential for approximating communication traffic. However, a significant number of systems - particularly in area-constrained edge devices - use bus-based interconnects. Bus-based systems cannot tolerate significant compression/decompression latency, requiring the use of different techniques.

Complementary to previous efforts, we explore approximate communication traffic compression in the context of bus-based systems. We propose a new, lightweight, approximate communication traffic compression scheme that is suitable for the tight latency constraints of bus architectures. Since the proposed hardware changes are limited to the bus interfaces, our proposal is applicable to systems that utilize pre-designed hard IP blocks, and to both on-chip and off-chip buses.

The key contributions of our work are as follows:

- We propose AxBA, an approximate bus architecture framework that enables transparent approximate compression of data transferred across buses to improve energy and performance.
- We propose a compression technique based on approximate deduplication, which is both lightweight and quality-configurable, *i.e.*, can conform to specified error constraints.
- We develop a software interface that application programs can use to specify regions in the system address space that can be subject to approximate compression, and a quality monitoring framework to dynamically modulate the error constraints for each region.
- We implement a prototype AxBA-based SoC using the Intel Nios II processor on a Cyclone-IV FPGA. Our evaluations on a suite of machine learning benchmarks reveal an average improvement in system performance of 29% and a 25% reduction in system-level energy for a 0.5% loss in application-level quality.

2 BACKGROUND AND MOTIVATION

Approximate compression reduces bus traffic and is applicable to a variety of bus topologies, *e.g.* on-chip, off-chip, serial, parallel, *etc.* For our discussion, we focus on parallel on-chip buses.

Parallel on-chip buses. Parallel on-chip buses are often used to integrate various components (*e.g.*, processors, accelerators, DMA controllers, peripherals and on-chip memory) in a System-on-Chip (SoC). These components can be classified into bus *masters*, which can initiate transactions on the bus, and bus *slaves*, which can only respond to transactions on the bus. If a bus contains multiple masters, it is said to be a *shared* bus and it must use an arbiter to grant access to the shared resources. Transferring data across parallel on-chip buses can be a major bottleneck for modern data-intensive workloads, especially with multiple masters trying to use a shared bus concurrently, causing *bus contention*. We focus on reducing this bottleneck by compressing the data traffic on the bus.

Motivation. Figure 1 motivates the opportunity for approximate compression of bus traffic by showing a trace of the values observed on the data bus over time for an eye detection application executing on a simple SoC consisting of an Intel Nios II processor, memory controller and peripherals. The colored horizontal bars represent values observed on the data bus in terms of how similar they are to the most frequently observed value, *viz.* 12. For example, the dark red bar indicates words on the bus that are exactly equal to 12, the light orange bar denotes values within an absolute difference of 1, *etc*. We observed that the fraction of exactly identical values on the bus is very small (4.5%). However, a significantly higher fraction of values are approximately similar or within a low dynamic range (53% within an absolute difference of 1 and 65% within an absolute difference of 2). In addition, most of these values appear in bursts, suggesting high compressibility. These observations underscore the significant opportunity available for reducing bus traffic using approximate compression. However, an approximate bus traffic compression scheme must: (i) be aware of which memory transactions may be approximated and which ones may not, (ii) provide an ability to control the error introduced by approximations, (iii) add minimal latency to bus transactions (so as to minimally impact system performance) and (iv) impose minimal effort on designers. AxBA addresses these challenges by proposing a lightweight, quality-aware compression scheme based on approximate deduplication, utilizing wrappers around existing bus masters and slaves to transparently compress/decompress data transactions, and introducing a software interface and runtime quality monitoring framework to facilitate software development.

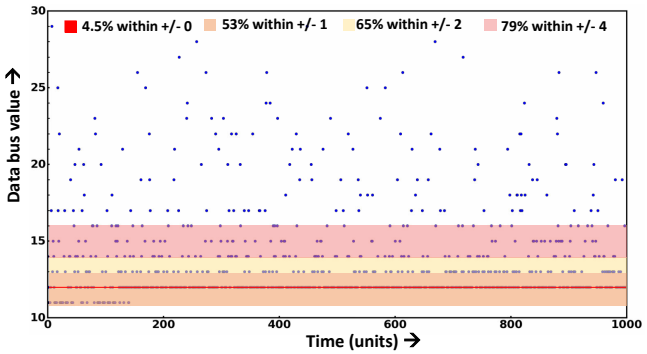

Figure 1: Data bus trace for an eye detection application

3 APPROXIMATE BUS ARCHITECTURE FRAMEWORK

Figure 2 provides a high-level overview of the Approximate Bus Architecture (AxBA) framework along with the proposed hardware and software enhancements. AxBA uses generic wrapper modules around existing masters and slaves that seamlessly enable compression and decompression of different regions in the system address space using different error constraints[1]. An AxBA module transparently intercepts a bus transaction and compresses/decompresses approximation-resilient data on the bus while adhering to the desired error constraint. It also transforms the original bus protocol to an AxBA bus protocol, which is virtually identical to the original bus protocol, with the exception of an additional control signal. The software enhancements consist of a programmer interface to specify regions in the address space that are amenable to approximations and a runtime quality monitoring framework that appropriately modulates the accuracy constraints for each of these regions to conform

[1]The master and slave components may be oblivious to compression, enabling predesigned components to be used without hardware changes.

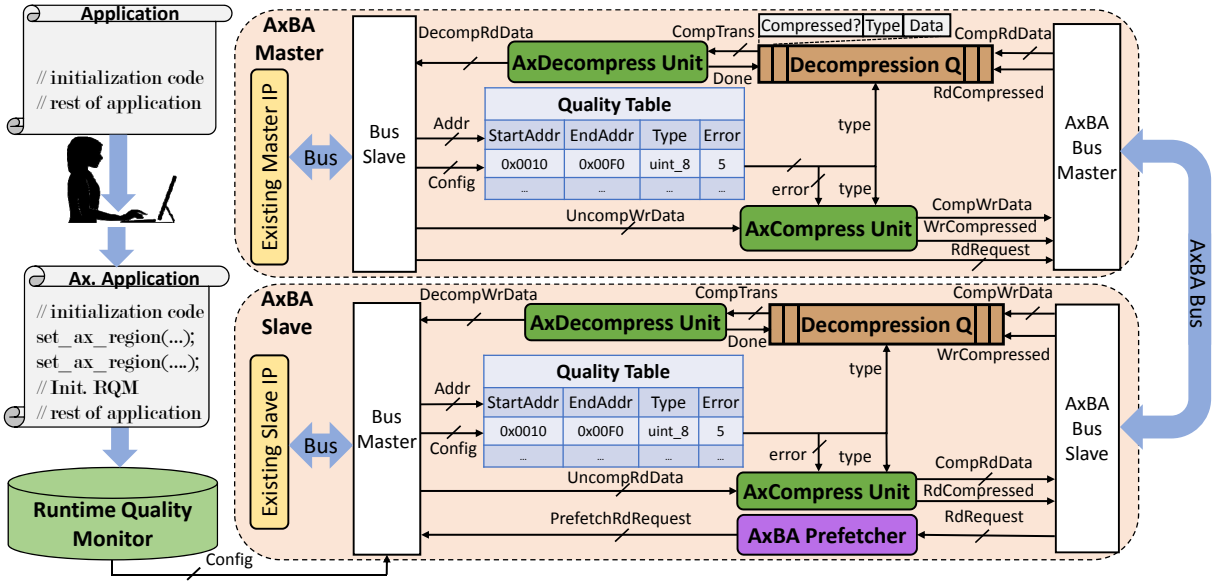

Figure 2: Overview of the AxBA framework

to a target application-level output quality. In this section, we discuss the approximate compression scheme used in AxBA, followed by a description of the specific hardware and software enhancements.

3.1 Approximate Compression Techniques

Approximate compression techniques for bus architectures should have the following desiderata: (i) quality configurability, (ii) low compression/decompression latency and (iii) minimal hardware complexity and overheads. Towards this objective, we explore two different lightweight approximate compression techniques, *viz.*, *Approximate Base-Delta (AxB+Δ) Compression* and *Approximate Deduplication (AxDeduplication)*. In each scheme, an accuracy constraint specifies the maximum error that may be incurred in compressing each data element. Further, the compression is constrained such that n data transactions on the bus are compressed to m transactions, where n and m are integers and $n > m$. This greatly reduces the complexity of the compression/decompression logic since there is no need to split (and eventually reconstruct) bus transactions. In the following paragraphs, we describe the two techniques in detail and explain the rationale behind the choice of Approximate Deduplication as the compression scheme in AxBA.

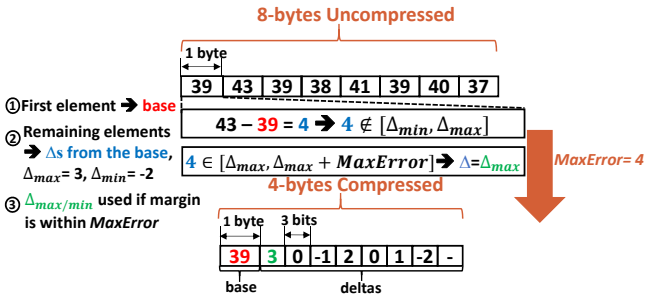

Figure 3: Approximate base-delta compression scheme

Approximate Base-Delta Compression. Base-Delta (B+Δ) [16], a lossless cache compression algorithm, exploits the low dynamic range of values in a data block to represent it with a common *base* and a series of *deltas*, *i.e.*, the differences between values within the block and the base. Since the deltas require fewer bits than the original values, this representation inherently compresses the original block. Inspired by this approach, we propose AxB+Δ where we subject the delta values to approximations. This is achieved by allowing an element to be represented by the maximum (minimum) delta value if it is within a certain accuracy bound from the maximum (minimum) delta value, thus extending the dynamic range of the scheme. Figure 3 shows an example 8-byte uncompressed data stream on the bus (equivalent to two bus words for a 32-bit data bus). The sequence consists of eight 1-byte signed integers. Let us assume that the maximum error magnitude that can be tolerated in compressing each data element is 4. To reduce the complexity of compression logic, the proposed approximate compression technique chooses the first element as the common base and analyzes the remaining values to determine the deltas while satisfying the desired accuracy constraint. The number of bits required to represent the delta values (3 bits in this example with $\Delta_{max} = 3$ and $\Delta_{min} = -2$) is precomputed based on the total uncompressed size of the data stream, the data bus size, and the size of the individual elements within the data stream. Using AxB+Δ, the seven remaining 1-byte elements (excluding the first base element) can be compressed into 3-bits each, with an additional byte for the base element. Thus, the entire data stream can be compressed to 4 bytes instead of the original 8 bytes, saving the transfer of a bus word, *i.e.*, 4 bytes.

To decompress a compressed data transaction, AxB+Δ adopts a similar approach to B+Δ, wherein the element values are obtained by simply adding the base value and the corresponding deltas.

Approximate Deduplication. Deduplication is a popular data compression technique that eliminates duplicate copies in repeating data.

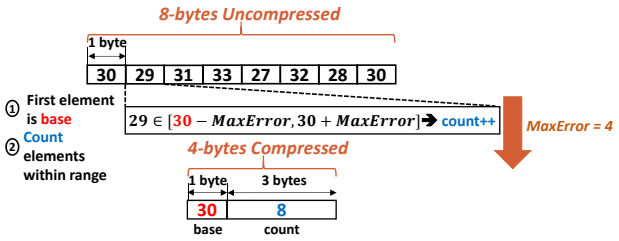

Figure 4: Approximate deduplication scheme

It has been widely used in secondary storage systems [11]. We propose a simple deduplication-based compression scheme inspired by run-length coding that can be realized with minimal hardware complexity. We store sequences of identical data values as a single value and its count. A compressed representation thus consists of a *base* element (the first element) and a *count* of how many times that element appears consecutively. *AxDeduplication* extends this compression scheme by relaxing the exact match requirement imposed for the consecutive elements. In other words, a value is interpreted as a repeated instance of the base value as long as the two differ by less than a specified error bound. Figure 4 shows an example 8-byte uncompressed data stream containing eight 1-byte signed integers, to which the proposed scheme is applied with a maximum permissible error magnitude of 4. The number of bits required to maintain the count is predetermined and is a function of the size of the elements in the data stream and the data bus width. In this example, the eight 1-byte elements can be compressed into a 1-byte base and a 3-byte count, resulting in a savings of 4 bytes. AxDeduplication can be further extended to store multiple repeating values along with their corresponding counts based on the bus width and the size of the underlying elements. For example, when compressing byte-sized scalar data transactions on a 4-byte wide bus, there are two other possible representations for a compressed transaction, *viz.*, two base elements and their repeating counts, or three base elements and their counts. These representations capture the higher likelihood of different repeating patterns in a data stream, while trading-off the maximum count that can be stored for each repeating value. Our experiments suggest that allowing two base elements greatly improves the compression ratio, while the additional benefits for three or more base elements are greatly diminished. Therefore, we choose a compressed representation with two base elements. To decompress a compressed transaction, we use the respective counts and repeat the corresponding base elements.

Comparison of Compression Techniques. In order to determine the compression technique for AxBA, we developed a software implementation for the two proposed schemes and compared them with a previously proposed low-overhead Bi-directional Precision Scaling scheme (BPS) [17]. Figure 5 quantifies the potential for reduction in communication traffic by comparing the compression ratios achieved across a suite of machine learning benchmarks (benchmark details are provided in Section 4) while ensuring that the application quality is maintained within 0.5%. We observe that AxDeduplication achieves much higher compression ratios compared to the other schemes. This is because in both BPS and AxB+Δ, each scalar element transmitted on the bus explicitly requires at least one bit for representation (either its truncated value or its delta from the

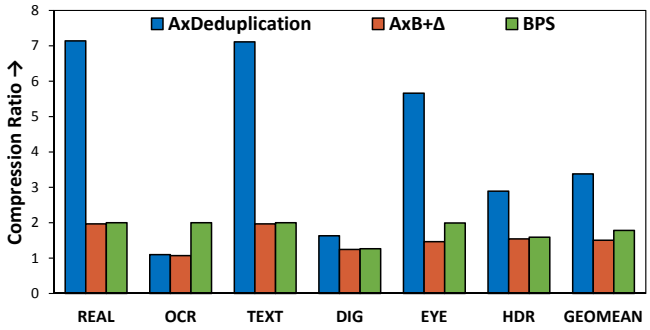

Figure 5: Comparison of compression techniques with 0.5% loss in application-level quality

base). Thus, the number of elements that can be represented depends linearly on the number of available bits in the compressed representation. However, AxDeduplication requires only one element to be represented explicitly, with the remainder being represented by a count. Therefore, in the limit, an exponential number of elements can be represented using the same number of bits, thereby offering higher opportunity for compression. Due to its potential for higher compression with minimal hardware complexity, we choose AxDeduplication as the compression scheme for AxBA. It is important to note, however, that the choice of approximate compression technique is not limited to those explored in this section; any low-latency, quality-aware compression technique can be implemented within the AxBA framework.

3.2 Hardware Support for AxBA

Figure 2 presents the design of a generic AxBA wrapper to realize approximate compression of communication traffic for a master/slave pair. An AxBA wrapper consists of: (i) A *Quality table* that captures the approximation-tolerant regions in the system address space and the error constraints associated with each region, (ii) An *AxCompress/AxDecompress unit* pair that perform the compression and decompression at runtime, and (iii) An *AxBA prefetcher* (specific to slaves) that prefetches requests from the slave to perform compression during reads at the slave end. An AxBA bus is identical to the underlying bus protocol, with an additional bit to indicate whether a transaction is compressed. The following paragraphs describe these components in greater detail.

Quality Table. In order to perform quality-aware compression, AxBA requires a mechanism to identify the bus transactions that can tolerate approximations. We propose the use of a fully-associative quality table that is programmable at runtime for this purpose. It consists of a small number of entries (4 in our experiments) with each entry identifying a region of the system address space that is amenable to approximation. Specifically, as shown in Figure 2, each entry includes: (i) the range of addresses that constitute the region, (ii) the data type of the elements in the region (*e.g.*, signed 16-bit, unsigned 8-bit, *etc.*), and (iii) the quality constraint (*e.g.*, maximum error magnitude) for the elements within the region. On each write request (read response) from the master (slave), the address of the transaction is compared with all the address ranges in the quality table to determine whether the address lies within any of the ranges (*i.e.*, whether the address has been marked as resilient to

approximations). If the address falls within a range in the table, the matching data type and error constraint are used by the Approximate Compress Unit to try to compress the data transaction. Otherwise, the transaction remains uncompressed for a miss. Similarly, for a read response (write request) in the master (slave), the quality table provides the data type of the elements in the compressed transaction, which is used to decompress this transaction. Given the small size of the quality table, our experiments indicate that it incurs negligible energy and performance overheads. Since the lookup is conducted in parallel across entries, there is minimal impact on latency.

Approximate Compression Unit. The Approximate Compression Unit (ACU) utilizes the proposed AxDeduplication scheme to dynamically compress an incoming read/write transaction. The ACU takes the error constraint and the data type from the quality table, and the uncompressed write data (read data) from the bus master (bus slave), to produce compressed write data (read data) and a control signal to indicate whether the transaction is compressed, *i.e.*, $WrCompressed$ ($RdCompressed$). Specifically, it compares each element in the uncompressed transaction with the earlier identified base element(s) to determine the corresponding repeating count(s) while ensuring that the error constraint is not violated. Depending on the achieved compression ratio, the ACU then packs multiple contiguous transactions into one (or more) compressed transaction(s). Note that the ACU does not finish compression until it receives a transaction in which the error constraint is violated, marking the end of the compressed transaction(s). However, since this process of ending one series of compressed transactions and beginning another is pipelined, the ACU incurs only one cycle of latency overhead per stream of transactions (*i.e.*, n transactions can be transmitted in n+1 cycles).

Approximate Decompression Unit. The Approximate Decompression Unit (ADU) receives a compressed transaction ($CompTrans$) from the Decompression Queue, which contains the data type of the elements comprising the compressed transaction, the actual compressed data, and a control signal indicating if the data is compressed. Accordingly, the ADU in the AxBA master (AxBA slave) decodes the repeating count(s) and the base value(s) embedded in *CompRdData* (*CompWrData*) and decompresses it to multiple transactions. This is achieved by repeating each base element an appropriate number of times, requiring a variable number of cycles. Note that even though the ADU incurs variable decompression latency, it still keeps the bus occupied by producing one decompressed transaction for the master (slave) in each cycle.

Decompression Queue. To hide the decompression latency of the ADU from the AxBA bus and remove the quality table lookup for decompression from the critical path, we introduce a Decompression Queue between the AxBA bus and the ADU. The queue temporarily holds the incoming AxBA bus transactions. When an AxBA bus transaction is ready, the queue stores *CompRdData* (or *CompWrData*), along with its data type (from the quality table). It then provides this information ($CompTrans$) to the ADU whenever the ADU finishes its current compressed transaction. We provision the size of queue to be the maximum number of data transactions that can be compressed into a single compressed transaction (8 in our implementation) to avoid any buffer stalls.

AxBA Prefetcher. A read request to an AxBA slave requires multiple read data responses from the underlying slave to perform compression. To this end, we introduce a simple next sequential prefetcher in the AxBA slave that initiates prefetch requests to the slave for each AxBA master read request. When an AxBA slave initially receives an incoming AxBA *RdRequest*, it first fetches data corresponding to the request address. The AxBA prefetcher then initiates multiple sequential prefetch requests ($PrefetchRdRequest$) so that the AxBA slave generates a compressed bus transaction. After the initial request is serviced by the AxBA slave, the AxBA prefetcher continues to speculatively generate prefetch requests to the slave for consecutive addresses until the ACU produces another compressed bus transaction ready to be consumed by the next request. If the next incoming AxBA bus request is sequential, *i.e.*, the request address matches the previously prefetched address, the AxBA prefetcher immediately services the request with the ready compressed bus transaction and begins generating speculative prefetch requests for the next request again. Otherwise, it clears the ready transaction and responds as if the incoming AxBA *RdRequest* was an initial request.

AxBA Bus. AxBA requires an additional 1-bit control signal that denotes whether an incoming transaction to AxBA master/slave is compressed. This necessitates enhancements to the standard bus interconnect including bus components such as arbiters, bridges, *etc* [2]. Note that it may be possible to reuse existing bus control signals for this purpose, but such optimizations depend on the specifics of the underlying bus protocol. In our experiments, we assume an additional control bit and account for the associated overheads.

3.3 Software Support for AxBA Framework

To minimize programmer burden, we propose a software interface that allows programmers to manually identify known approximation-tolerant regions in the system address space, and propose a runtime quality monitoring (RQM) framework that automatically obtains error constraints for the identified regions while ensuring that the target application quality is satisfied. The following paragraphs describe them in detail.

Software Interface. We introduce a function (`set_ax_region`, shown in Figure 2) that a programmer uses to manually identify approximation-tolerant regions in the system address space. Specifically, it creates an entry in the quality table that is exposed to AxBA through memory-mapped registers.

Runtime Quality Monitoring Framework. RQM requires the programmer to specify: (i) the approximation-tolerant regions, (ii) the application-level quality constraint, and (iii) a quality function to evaluate output quality. It then executes the application in two interleaved phases – *calibration phase* and *evaluation phase*. During the calibration phase, RQM obtains the error constraints for the programmer-identified regions. The application is then executed with these constraints in the evaluation phase. The calibration phase is invoked at regular intervals to evaluate the programmer-specified quality function. In this phase, RQM re-executes the application for

[2]These enhancements must be done for each unique underlying bus protocol AxBA is used with

each input with no approximation, *i.e.*, error constraint of 0, and subsequently evaluates the quality function using the outputs with and without approximation. In case the evaluated output quality satisfies (violates) the application-level constraint, RQM greedily relaxes (tightens) the error constraint for the most communication-intensive region. The communication intensity for an approximation-tolerant region is obtained using access counters embedded in the AxBA wrappers.

4 EXPERIMENTAL METHODOLOGY

In this section, we describe the experimental setup and the benchmark applications used to evaluate AxBA.

Experimental Setup. We designed two prototype systems using a Cyclone-IV FPGA development board — a single-core system and a multi-core system. The single-core system consists of a Nios II processor, an on-chip scratchpad, a DMA engine, and an SDRAM controller interfaced with an on-board SDRAM. The multi-core system (shown in Figure 6) consists of two Nios II cores, each with a private on-chip scratchpad and a DMA engine, interfaced to the SDRAM through a shared SDRAM controller. To realize approximate compression of bus traffic, we introduced AxBA wrappers around the DMA engine and the SDRAM controller for both the systems. This resulted in an area overhead of 7.7% for both the multi-core and single-core systems. The max frequency (Fmax) worsened by 2.3% in the multi-core system, while the single-core system incurred no delay overhead. To obtain energy estimates, we used Altera's PowerPlay Power Analyzer tool with a 50 *ms* simulation trace that captures the key computational kernels for each application. We utilized a performance counter in the Nios II core to measure the execution time for each benchmark.

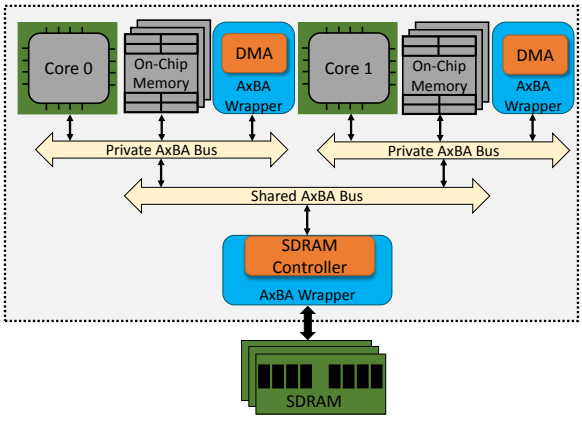

Figure 6: Test system used in our evaluation

Benchmarks. Table 1 lists the benchmarks and the datasets used used in our experiments. The quality metric for each benchmark is also provided. Each of these benchmarks were executed in isolation on the single-core system, and in a data-parallel fashion on the multi-core system.

5 EXPERIMENTAL RESULTS

5.1 Single-core performance and energy benefits

Figure 7 shows the normalized execution time of the benchmarks when using the proposed AxBA framework in a single-core system.

Table 1: Machine Learning Benchmarks

Application	Algorithm	Dataset	Quality metric
Handwritten Digit Recognition (HDR)	Support vector machines	MNIST	Classification accuracy (fraction of inputs correctly classified)
Text Classification (TEXT)		REUTERS	
Character Recognition (OCR)	K-nearest neighbors	OCR digits	
Eye Detection (EYE)	Generalized learning vector quantization	YUV faces	
Handwritten Digit Classification (DIG)	K-nearest neighbors	Gisette	
Document Clustering (REAL)	K-means clustering	Real-sim	Mean cluster radius

The execution times are normalized to a baseline design which does not use bus traffic compression. As shown in the figure, AxBA results in an execution time reduction of up to 42% (average of 19%) with an average 0.5% loss in application-level accuracy. This is mainly due to the reduction in bus traffic with approximate compression, resulting in a reduced volume of data being transferred across the bus.

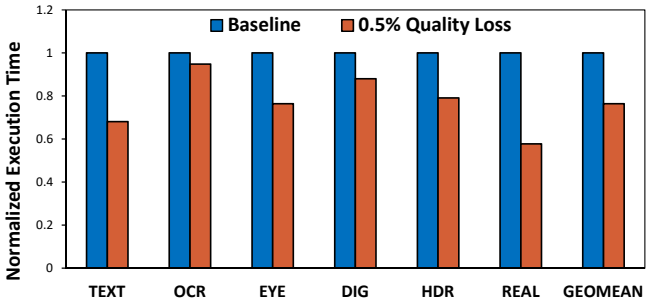

Figure 7: Single-core execution time benefits with AxBA

Figure 8 demonstrates the system-level energy benefits achieved by the AxBA framework. In the isolated system, energy benefits range from 3% to 41% (average of 15%) while incurring negligible loss (average 0.5%) in output quality. The energy benefits are directly proportional to the reduction in execution time after accounting for the additional hardware overheads in AxBA.

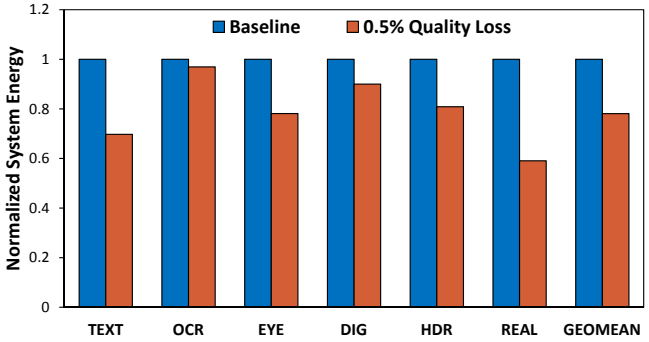

Figure 8: Single-core system energy benefits with AxBA

5.2 Dual-core performance and energy benefits

Figure 9 illustrates the normalized execution time of the benchmarks for an application-level output quality constraint of 0.5% using the

proposed AxBA framework when running the dual-core system in a data-parallel fashion. As seen in the figure, the AxBA framework results in a reduction in execution time of up to 50% (average of 29%) with a small loss in application-level accuracy (0.5%). The execution time benefits are higher than for the single-core system because in this case AxBA also minimizes contention for the shared bus.

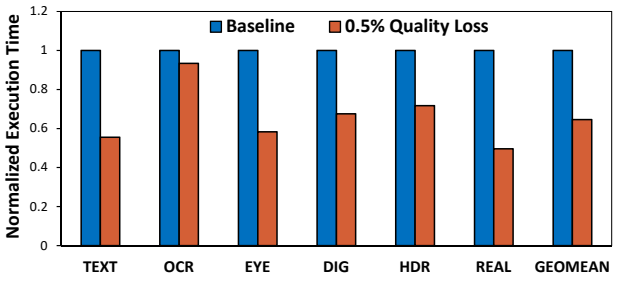

Figure 9: Congested system execution time benefits with AxBA

Similarly, in Figure 10, we show the system-level energy benefits for the the multi-core system. In this case, the AxBA framework achieves benefits in system-level energy ranging from 3.7% to 49%, (average of 25%).

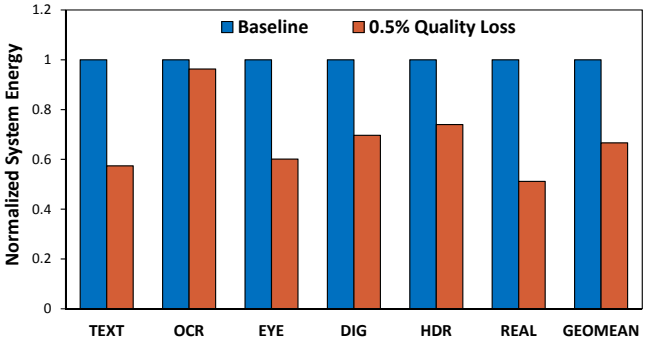

Figure 10: Congested system energy savings with AxBA

5.3 Compression ratio *vs.* quality trade-off

Figure 11 demonstrates the compression ratio *vs.* application-level quality trade-off obtained by varying error bounds for the data structures that are amenable to approximations. The results are presented for two representative benchmarks, EYE and HDR. The figure also shows the compression ratios observed for an error bound of 0 (*i.e.*, lossless compression). In both cases, we achieve higher compression ratios at negligible output quality loss with more relaxed data structure error bounds. Specifically, HDR exhibits a slow and steady increase in compression ratio along with a smaller loss in application quality as the error bound is relaxed. On the other hand, for the EYE application, we observe a steeper increase in the compression ratio as the permissible error increases, but the accompanying increase in application-level quality loss is also much steeper.

5.4 AxBA in action: Case study

Figure 12 visually illustrates AxBA in action by plotting the bus traffic observed during a 30 *ms* time window with the HDR application executing on the single-core system, as well as the bus traffic

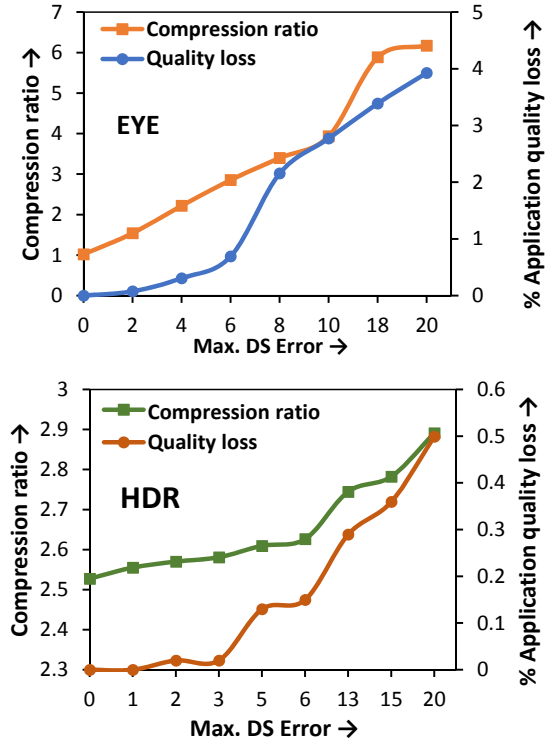

Figure 11: Compression ratio *vs.* output quality trade-off for two representative benchmarks

observed in the baseline system for the same duration. Notice that the traffic on the bus in both cases is fairly regular and interleaved with the computation phases of the application. This is because the system has a software-managed scratchpad interfaced with the processor that requires explicit DMA transfers to copy the data from the off-chip SDRAM to the scratchpad memory (and vice versa). As shown in the figure, AxBA significantly reduces the data transfer time during each phase, which eventually leads to improved system performance. Note that the time spent in the computation phases remains identical for both the systems, and the performance benefits are primarily due to the reduced volume of data that is transferred across the bus.

6 RELATED WORK

In this section, we present an overview of the related efforts in approximate communication, and highlight the distinguishing features of our work. Previous works in approximate communication can be classified based on the type of interconnects (off-chip *vs.* on-chip) and the nature of communication traffic they target.

Approximate memory compression. [8] proposed extensions to the load/store queue for handling variable precision across the compute and memory systems, allowing for more concise representation in memory. [20] designed a cache architecture that stores similar valued cache lines as a single cache line, increasing the effective cache capacity. [17] proposed a quality-aware memory controller that transparently compresses/decompresses off-chip memory traffic. All these efforts focus solely on the memory traffic, and do not target

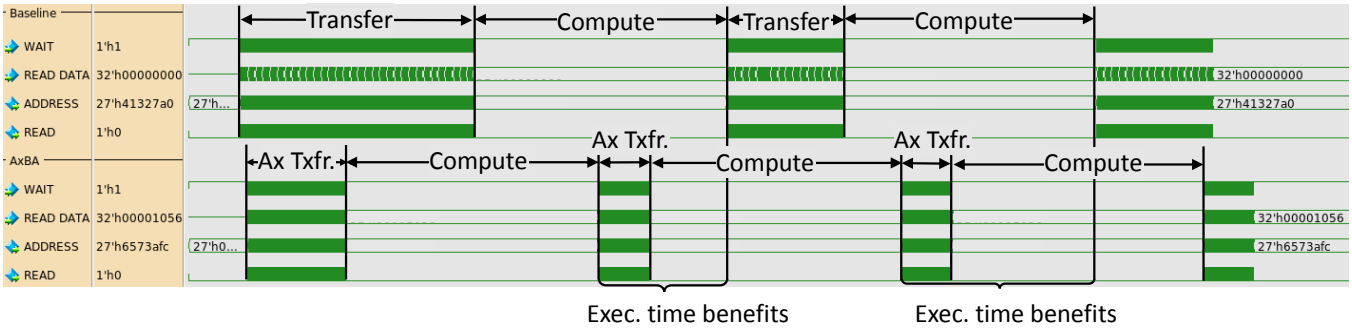

Figure 12: Bus traffic over time with single-core execution of handwritten digit recognition application

other communication traffic. In addition, these techniques also require significant changes to existing IPs, *i.e.*, processor cores, cache hierarchy, memory controller *etc*. In contrast, AxBA focuses on the entire communication traffic and requires no changes to the existing bus masters and slaves.

Approximate off-chip communication. In the context of off-chip communication, [7, 10, 15] propose approximation techniques to reduce data transitions on the off-chip serial bus, thereby reducing bus energy. However, unlike our proposal, these efforts do not reduce the overall bus traffic, and hence do not offer any improvements in performance.

Approximate on-chip communication. In the context of on-chip communication, [12] proposes the use of a lightweight and lossy NoC for latency-sensitive packets, *e.g.*, critical word response for cache misses, in conjunction with a regular NoC for the other packets to improve execution time without impacting application quality. This is orthogonal to AxBA, which focuses on mitigating the overall impact of communication traffic, and not just the critical word latency. [2] focuses on reducing communication traffic in NoCs using frequent pattern and dictionary-based approximate compression techniques. These compression techniques are simply not applicable to bus-based architectures that impose stricter latency constraints, requiring the exploration of new lightweight approximate compression techniques for AxBA. Moreover, our proposal is applicable to both on-chip and off-chip bus-based architectures.

7 CONCLUSION

This paper presents an approximate bus architecture framework, AxBA, that leverages the intrinsic error resilience of modern workloads to reduce communication traffic, improving both energy and performance. The AxBA framework transparently compresses/decompresses transactions within approximation-resilient regions of the system address space, and requires no changes to existing bus masters and slaves. We propose a lightweight compression scheme, Approximate Deduplication, that is well-suited for use within the AxBA framework. We provide a software interface to AxBA and a runtime quality monitoring framework that minimizes programmer effort for an AxBA-based system. An FPGA prototype is used to demonstrate the energy and performance improvements realized by AxBA.

REFERENCES

[1] L. Benini, D. Bruni, A. Macii, and E. Macii. 2002. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proc. of DATE*.

[2] R. Boyapati, J Huang, P. Majumder, K.H. Yum, and E.J. Kim. 2017. APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures. In *Proc. of ISCA*.

[3] R. Canal, A. Gonzalez, and J.E. Smith. 2000. Very low power pipelines using significance compression. In *Proc. of MICRO*.

[4] V. K. Chippa, D. Mohapatra, K. Roy, S.T. Chakradhar, and A. Raghunathan. 2014. Scalable Effort Hardware Design. *IEEE Trans. on VLSI Systems* (Sept 2014).

[5] D. Citron and L. Rudolph. 1995. Creating a wider bus using caching techniques. In *Proc. of HPCA*.

[6] M. Farrens and A. Park. 1991. Dynamic base register caching. In *Proc. of ISCA*.

[7] D. Jahier Pagliari, E. Macii, and M. Poncino. 2016. Approximate Differential Encoding for Energy-Efficient Serial Communication. In *Proc. of GLSVLSI*.

[8] A. Jain, P. Hill, S.C. Lin, M. Khan, M.E. Haque, M.A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. 2016. Concise Loads and Stores: The Case for an Asymmetric Compute-Memory Architecture for Approximation. In *Proc. of MICRO*.

[9] S. Jain, S. Venkataramani, and A. Raghunathan. 2016. Approximation through Logic Isolation for the Design of Quality Configurable Circuits. In *Proc. of DATE*.

[10] Y. Kim, S. Behroozi, V. Raghunathan, and A. Raghunathan. 2017. AXSERBUS: A Quality-Configurable Approximate Serial Bus for Energy-Efficient Sensing. In *Proc. of ISLPED*.

[11] P. Kulkarni, F. Douglis, J. LaVoie, and J.M. Tracey. 2004. Redundancy Elimination Within Large Collections of Files. In *Proc. of USENIX*.

[12] Z. Li, J. San Miguel, and N. Enright Jerger. 2016. The runahead network-on-chip. In *Proc. of HPCA*.

[13] C. Liu, A. Sivasubramaniam, and M. Kandemir. 2004. Optimizing bus energy consumption of on-chip multiprocessors using frequent values. In *Proc. of Euromicro PDP*.

[14] S. Liu, K. Pattabiraman, T. Moscibroda, and B.G. Zorn. 2011. Flikker: Saving DRAM Refresh-power through Critical Data Partioning. In *Proc. of ASPLOS*.

[15] D. Jahier Pagliari, E. Macii, and M. Poncino. 2016. Serial T0. In *Proc. of DAC*.

[16] G. Pekhimenko, V. Seshadri, O. Mutlu, M.A. Kozuck, P.B. Gibbons, and T.C. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proc. of PACT*.

[17] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan. 2017. Approximate Memory Compression for Energy-efficiency. In *Proc. of ISLPED*.

[18] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. 2014. ASLAN: Synthesis of Approximate Sequential Circuits. In *Proc. of DATE*.

[19] A. Ranjan, S. Venkataramani, Z. Pajouhi, R. Venkatesan, K. Roy, and A. Raghunathan. 2017. STAxCache: An approximate, energy efficient STT-MRAM cache. In *Proc. of DATE*.

[20] J. San Miguel, J. Albericio, A. Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: A Cache for Approximate Computing. In *Proc. of MICRO*.

[21] S. Sardashti, A. Arelakis, P. Stenström, and D.A. Wood. 2015. *A Primer on Compression in the Memory Hierarchy*. Morgan & Claypool Pubs.

[22] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. 2011. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. In *Proc. of ESEC/FSE*.

[23] S. Venkataramani, S.T. Chakradhar, K. Roy, and A. Raghunathan. 2015. Approximate Computing and the Quest for Computing Efficiency. In *Proc. of DAC*.

[24] S. Venkataramani, V.K. Chippa, S.T. Chakradhar, K. Roy, and A. Raghunathan. 2013. Quality Programmable Vector Processors for Approximate Computing. In *Proc. of MICRO*.

[25] P. Zhou, B. Zhao, Y. Du, Y. Xu, Y. Zhang, J Yang, and L. Zhao. 2009. Frequent Value Compression in Packet-based NoC Architectures. In *Proc. of ASP-DAC*.