An Overflow-free Quantized Memory Hierarchy in General-purpose Processors

Marzieh Lenjani[1], Patricia Gonzalez[2], Elaheh Sadredini[1], M Arif Rahman[1], Mircea R. Stan[2]
[1]*Department of Computer Science,* [2]*Department of Electrical & Computer Engineering*
University of Virginia
Charlottesville, VA USA
{ml2au, lg4er, elaheh, mir6zw, mircea}@virginia.edu

Abstract*—Data movement comprises a significant portion of energy consumption and execution time in modern applications. Accelerator designers exploit quantization to reduce the bitwidth of values and reduce the cost of data movement. However, any value that does not fit in the reduced bitwidth results in an overflow (we refer to these values as outliers). Therefore accelerators use quantization for applications that are tolerant to overflows. We observe that in most applications the rate of outliers is low and values are often within a narrow range, providing the opportunity to exploit quantization in general-purpose processors. However, a software implementation of quantization in general-purpose processors has three problems. First, the programmer has to manually implement conversions and the additional instructions that quantize and dequantize values, imposing a programmer's effort and performance overhead. Second, to cover outliers, the bitwidth of the quantized values often become greater than or equal to the original values. Third, the programmer has to use standard bitwidth; otherwise, extracting non-standard bitwidth (i.e., 1-7, 9-15, and 17-31) for representing narrow integers exacerbates the overhead of software-based quantization. The key idea of this paper is to propose a hardware support in the memory hierarchy of general-purpose processors for quantization, which represents values by few and flexible numbers of bits and stores outliers in their original format in a separate space, preventing any overflow. We minimize metadata and the overhead of locating quantized values using a software-hardware interaction that transfers quantization parameters and data layout to hardware. As a result, our approach has three advantages over cache compression techniques: (i) less metadata, (ii) higher compression ratio for floating-point values and cache blocks with multiple data types, and (iii) lower overhead for locating the compressed blocks. It delivers on average 1.40/1.45/1.56× speedup and 24/26/30% energy reduction compared to a baseline that uses full-length variables in a 4/8/16-core system. Our approach also provides 1.23× speedup, in a 4-core system, compared to the state of the art cache compression techniques and adds only 0.25% area overhead to the baseline processor.

I. Introduction

Data transfer across the memory system and interconnect constitute a significant fraction of the total energy and performance in memory-intensive applications [1], [2], [3], [4], [5], [6]. Prior works [7] show that the energy cost of fetching a 32-bit word of data from off-chip DRAM is 6400× higher than an ADD operation. This trend worsens as the processor technology moves to smaller nodes.

Prefetching and forwarding techniques [8] can alleviate the performance cost but they can not reduce the energy cost of data movements. Therefore, several approaches proposed to trade off the accuracy for the size of transferred data by omitting (truncating) a certain number of least significant bits (LSBs) in the mantissa of floating-point numbers [9], [10]. We refer to these methods as OLSB. Two factors limit the benefit of such techniques: (i) the overhead of eight bits for the exponent, and (ii) the high error that grows with the value of the exponent. In other words, the larger the magnitude of the value, the higher the absolute error (the magnitude of error). Despite the unfavorable effects, the floating-point format and the exponent part are necessary for supporting a wide range of values. In fact, the floating-point format is popular among developers because it supports a wide range of values and decreases the probability of overflow during arithmetic operations. Due to this popularity, processor vendors added floating-point ALUs as soon as there were enough transistors available on the chip.

Our characterization demonstrates that, in real applications, most of the data values lie within a limited range and only a small fraction of data values are located at the tail of the distribution where these values are relatively further from the average. We define these values as *outliers*. Based on this observation, we make a case for using a specific type of *quantization* (biased fixed-point), as a mean to reduce the bitwidth of the variables and reduce the cost of data movement. This type of quantization maps a range of values to a set of discrete indexes and therefore it requires few bits to represent indexes [7], [11], [12], [13].

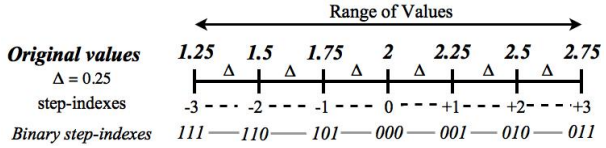


Fig. 1: Quantizing a range of values to 3-bit integers

Figure 1 shows that this type of *quantization* reduces the number of bits required for representing values in the range of 1.25 to 2.75 by dividing the range into six steps (Δ=0.25) and mapping them into 3-bit integers (*step-index*). In this method, outliers can significantly expand the range and consequently

increase the number of bits required to represent the quantized values, even though they are accessed very infrequently. A simple solution for handling outliers is to map values that result in overflows to the maximum or minimum of the range. The effect of such mapping depends on the application. For example, in BlackSholes, mapping outliers to the maximum or minimum of the range increases maximum absolute error by 116%, 333% and 69355% for bitwidth of 4, 8, and 12, respectively (as bitwidth increases our method's error decreases and hence the ratio of the error caused by outliers to our method's error increases).

A software implementation of quantization in general-purpose processors imposes a significant conversion overhead and programmer's effort, is prone to overflow, and more importantly, can not unlock maximum benefit for variables that require less bitwidth than standard variables. It has to use only 8-bit, 16-bit, or 32-bits variables and hence it imposes significant (e.g., 100%, 77%, or 88% for 4-bit, 9-bit, or 17-bit variables, respectively) cache space and memory bandwidth overhead (more details in Section II).

Due to these constraints, quantization is popular in accelerators, where the bitwidth can be customized [14], [15], [16]. However, having one accelerator for each application, especially in consumer devices, is impossible, due to space constraints, scheduling overheads, communication overheads and, interconnection limitations.

The goal of this work is to propose and evaluate an overflow-free and transparent architectural support for quantization in memory hierarchy of general-purpose processors to accelerate a large domain of memory-intensive applications, where variables can be represented with minimum and flexible number of bits. Our hardware modules act as accelerators for conversions that transparently quantize and dequantize cache blocks as they move between L1 and L2 (or alternatively between L2 and L3). Accordingly, the data values are quantized in the memory hierarchy in L2 and beyond and are dequantized when transferred to L1, saving the capacity of L2, L3, and memory as well as bandwidth of L3 and memory. The values are in their original format in L1 and hence quantization causes no overflow during computation. To prevent overflow while we quantize and transfer data to L2, we propose to store and represent outliers in a separate space, assigned to outliers, and propose a mechanism for retrieving these values.

Quantization could be considered as a specific type of compression. However, cache compression techniques impose a significant metadata overhead and need a complex mechanism for locating the address of compressed values in compressed caches (translating the address). Due to these overheads, most of the cache compression techniques are only amenable for L3 [17], [18] and not applicable for L2 (more details in Section II). We observed that the inefficiencies stem from the fact that compressor and decompressor have minimum information about the program and obliviously search for value locality within each cache block that they receive [17], [19]. We exploit the predictability of the range of values and fixed bitwidth in quantization and devised a software-hardware interaction to address inefficiencies in cache compression techniques. The interaction transfers specific characterization of applications such as data layout, distribution of values, and tolerable error (translated to *mid*, *step-size*, and *bit-width*) to hardware. Our hardware modules use this information to track which pages belong to which array of the application and hence store metadata only once for all pages of an array, reducing the metadata overhead. More importantly, our hardware modules exploit the bitwdith information for a light-weight address translation mechanism, implemented by arithmetic operations. This simplified address translation mechanism enables us to have a compressed L2 in addition to L3.

This paper makes the following contributions:

- We characterize 11 *real data sets* to show that real applications operate on data values within a particular narrow range, with only few outliers out of the range, suggesting that a significant portion of values can be represented by few bits.
- We propose efficient techniques to provide support for *quantization in hardware*. First, we propose a simple software-hardware interface to specify quantized variables and the necessary parameters. Second, we introduce efficient hardware modules that transparently quantize and dequantize cache blocks between a upper-level cache and a lower level cache. Third, we propose an efficient way of supporting outliers.
- Our evaluation of approximate applications shows that quantization provides on average 39-98% better accuracy compared to the techniques that omit the LSBs. Quantization provides on average a speedup of 1.40/1.45/1.56× and energy reduction of 24/26/30% compared to a baseline that uses full-length variables in a 4/8/16-core system. We have synthesized the RTL implementation of our hardware modules [20] and the synthesize report shows that our method adds only 0.25% area overhead to the baseline processor.

II. Motivation

In this section, we explain the benefit of hardware-based quantization over three alternative approaches: (i) quantization in software, (ii) OLSB, and (iii) cache compression.

A. Quantization in Hardware versus Software

A software-based implementation of the quantization method suffers from four disadvantages. First, it imposes the overhead of multiple instructions for each conversion. Figure 2 (a) demonstrates multiple examples of necessary conversion points: (i) quantization before storing values in arrays (❶ and ❸), (ii) dequantization before functions that requires the real values, such as sine and cosine (❷), (iii) dequantization before computation on non-quantized values (❹), (iv) conversion to avoid overflow (❺), and (v) dequantization before storing the final results in the output file (❻). Second, it is error-prone as programmers should manually detect the locations of necessary conversions. Third, when the required *bit-width* is 1-7, 9-15, 17-31, it uses 8-bit, 16-bit and 32-bit variables, respectively (to avoid the overhead of addressing and extracting a few

bits in a sequence of bits), imposing up to 700%, 77% and 88% overhead of cache space and memory bandwidth. Fourth, it cannot represent outlier values, which is quite common in real data sets (explained in Section III). For example, in Blackscholes, most price values can be represented by 6 bits but the maximum price requires 18 bits. With no support for outliers, software-based quantization has to use 32 bits for all values. We can implement our proposed method for outliers in software and store them in a separate array. However, in this case, quantization (❼) and dequantization (❽) functions becomes more complex as they have to check outliers and read/write outliers from/in a separate space(Figure 2 (b)).

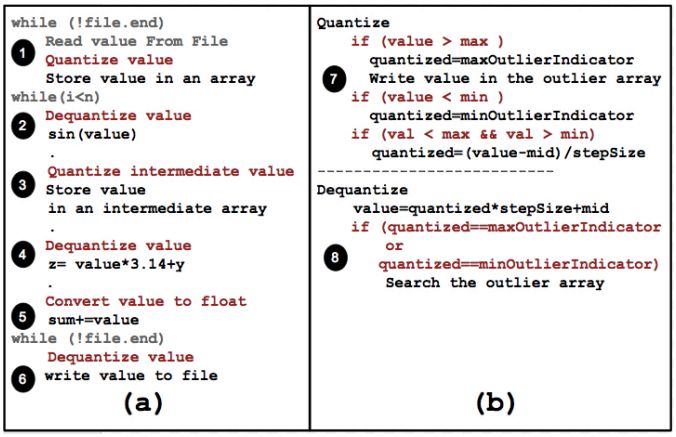
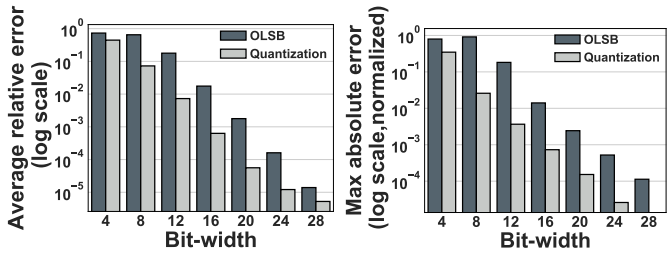


Fig. 2: Quantization in software

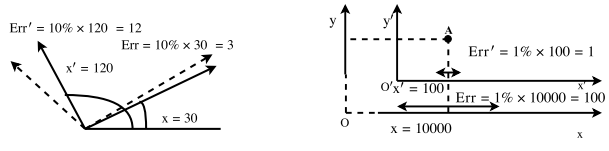
B. Quantization versus Omitting the LSBs.

This section discusses two major benefits of using quantization in approximate applications.



(a) Relative error (b) Absolute error

Fig. 3: Final output error using quantization vs. OLSB



(a) Angles (b) Coordinates

Fig. 4: Relative error for (a) angles, and (b) coordinates

***(i)* Lower relative error/shorter bit-width.** Figure 3 (a) compares the error introduced by quantization to the error introduced by OLSBs while varying the bit-width from 4 to 30 in eight popular approximate applications (details on the methodology is available in Section VI-A). For OLSB, we use one bit for the sign and the rest of the bits for mantissa (unless the bitwidth is 4 bits, where we use one bit for the sign and three bits for exponent and assume mantissa is one). The accuracy metric here is the *average relative error* (average of the percentage of error for all output variables), which has been used in prior works on approximation [9], [27], [28]. This figure clearly shows that, for any given bit-width, quantization's error is lower than OLSB's error. It also shows that, for the same level of error, quantization requires fewer bits. Shorter bit-width stems from the fact that, unlike OLSB, quantization does no need eight bits for the exponent part.

***(ii)* Lower absolute error/shorter bit-width.** When the accuracy metric is the relative error, the magnitude of the error can grow with the magnitude of the original value. However, many approximate applications expect that the magnitude of error remains limited regardless of the original value. For example, Inversek2j (an application from AxBench suit [27]), is an application that calculates the rotation angle for a 2-joint robotic arm. Assume that we define a relative error of 10% as the tolerable error. In this case, if the arm moves a small angle to hold the object, such as 30°, it will be off by only 3°, but for large angles, such as 120°, the error becomes 12°, which is quite high and can potentially make the arm miss the target object (Figure 4(a)). In reality, the acceptable error depends on the diameter of the target object, which is a fixed value and does not depend on the original value of the rotation. Another example is the inputs of Jmeint (from AxBench suit [27]) that analyzes the overlap of a pair of triangles in the 3-D space. In the real world, the acceptable error for the coordinates of the point A should not depend on the location of the center of the cartesian system (Figure 4(b)). For these applications, the absolute difference between the original value and the approximate value defines the proper accuracy metric. Unfortunately, when we omit the LSBs from the mantissa, the absolute error depends on the value of the exponent ($Error = (-1)^S \times \left(\sum_{i=23-(l+1)}^{23} M_{23-i}2^{-i}\right) \times 2^E$) which can lead to a high absolute error if the exponent value is large whereas quantization limits the maximum possible absolute error to the *step-size*. Figure 3 (b) demonstrates that, compared to OLSB, quantization lowers maximum absolute error in the output of our evaluated approximate applications when the *bit-width* is varied from 4 to 30.

C. Quantization versus Cache Compression.

Cache compression techniques have three problems. First, they require metadata per cache block. For example, Base-delta [19] shrinks the size of each cache block by subtracting the values within the block from a base value. It requires 1-4 bytes for the base value per block. For a compression ratio as high as four, the four bytes, per compressed block, imposes 25% overhead. Second, they can not achieve a reasonable compression rate for two types of arrays: (i) arrays containing single-precision floating point values, where variation in the least significant bits is high and (ii) arrays of structure or any other composite data type with consecutive variables that are inherently different. Third, cache compression techniques,

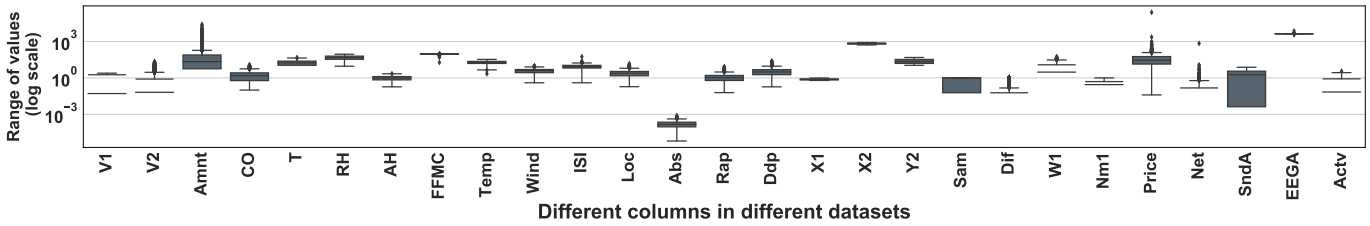
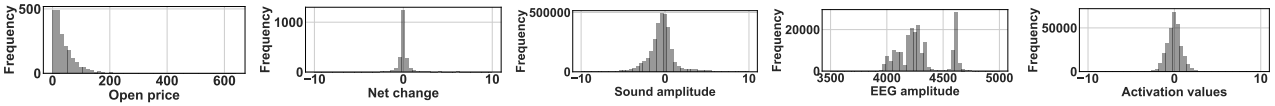


Fig. 5: Variables in real datasets exhibiting a limited range (details in Table I)



(a) Blksh, NYSE [21] (b) Financial, NYSE [21] (c) FFT, ASD [22], [23]) (d) FFT, EEG [24] (e) VGG, Cifar10 [25], [26])

Fig. 6: Histograms of data values in various applications

compress each block of the cache into different size which requires a complex mechanism for locating the cache blocks. As a result, they employ one of the three following techniques: (i) padding that pads compressed block so that the size of each compressed block becomes an integer and power of two fraction of the size of the original cache block (e.g. LCP [29]) (i) dividing the original cache block into integer number of segments and assigning one tag per each segment (e.g. HyComp [17]), (ii) employing a completely decoupled tag array and data array, where the tag array points to the start of the compressed block in the data array (e.g. Decoupled compressed cache [18]). The first and second approach constraint compression ratio and the third approach requires modification in the cache and a defragmentation mechanism for the data array. More importantly, tag and data can not be accessed in parallel and will be accessed sequentially, doubling the latency of caches (which is already around 36 cycles for modern large last level caches). Since the complex decoupling mechanism cannot be employed for L2 caches, most cache compression techniques only compress the last level cache. In quantization, the compressed bitwdith is fixed and the page offset of each compressed variable can be determined by arithmetic operations. Section IV-D explains that we exploit the fixed bitwidth to locate our compressed values using simple arithmetic operations and keep the original structure of the cache, eliminating the decoupling mechanism. This enables us to quantize values in L2 in addition to L3. More importantly, by transferring information about data layout to hardware, we track which pages belong to which array of the application and hence store metadata only once for all pages of an array.

III. Key Observations and Key Ideas

This section explains three key observations that form three key ideas of this paper.

Observation 1: Data Values in a narrow Range. We characterize 11 *real data sets* [21], [23], [24], [26], [30], [31], [32], [33], [34], [35] used in different domains, such as machine learning, weather forecast, financial analysis, signal processing, image recognition, etc. (details in Table I). Figure 5 illustrates the box plot for several variables in these datasets, where the box shows the range of values in the first to third quartile, the bars show the lower and upper limit within 1.5× of the first and third quartile, and the values outside that range are shown as dots. This figure clearly demonstrates that data sets for many approximate applications exhibit values within a narrow range. The first key idea is that, given the narrow range of values, quantization can be applicable to a wide range of applications and efficiently reduce the cost of data movement.

TABLE I: Description of real data sets

Variable name	Data set description
V1, V2, Amnt	Credit card fraud detection data set [30]
CO, T, RH, AH	Sensor data for air quality [31]
Temp, Wind, ISI	Weather index for forest fire [32]
Loc, Abs, Rap, Ddp	Speech data from parkinson patients [33]
X1, X2, Y2	Intrusion detection data set [34]
Sam, Dif	Building shapes for energy efficiency [35]
W1, Nm1	Purchase in sale transactions [36]
Price, Net	Stock exchanges [21]
SndA	Sound amplitudes [23]
EEGA	EEG amplitudes [24]
Actv (VGG)	Images for deep learning [26]

Observation 2: Outliers. Figure 5 also demonstrates that some values will be outside the limited range (outliers) and Figure 6 shows that data values in most real applications exhibit a normal/folded normal distribution. According to the normal distribution definition, 99.99% of the data values are within 8 standard deviation and only 0.0001 (0.01%) of the values fall outside this range. There are two common approaches to deal with outliers in data analysis techniques [37]: *(i)* mapping outliers to a minimum or maximum value, or *(ii)* processing the outliers with their original values if the outliers provide meaningful insight to the analysis [37], [38], [39], [40]. Our proposed method provides support for both approaches. The second key idea is to use the lowest possible number of bits for the most common values and store and rep-

resent outliers separately, for applications that require support for the second approach.

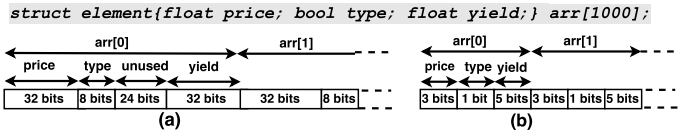


Fig. 7: (a) Original and (b) quantized array of structure

Observation 3: Common data layout in memory intensive applications. Pointers impose a significant overhead [41], [42], [43]. Consequently, memory intensive applications, with high spatial locality, layout data in two different ways: (i) array of structures (AoS), or (ii) structure of arrays (SoA). For example, in graph processing applications (which intuitively should use a linked list), we prefer arrays of edges and vertices or sparse matrices [44], partly, because pointer chasing, dynamic memory allocation (for each element of the data structures such as linked list) and additional random memory access of linked lists imposes a significant overhead. Supporting quantized SoA is straightforward as consecutive elements have same quantization parameters. However, providing support for AoS requires a complex metadata handling and address translation mechanism (Figure 7 shows how AoS should be quantized). Accordingly, the third key idea is communicating data layout to hardware and track which pages belong to which arrays to keep metadata once per whole array and use the layout information for simplifying address translation. Communicating the layout, also enables eliminating the unused bits before byte variables (used for alignment of composite data types, as shown in Figure 7), compressing boolean values (which only need one bit), and compressing integers within narrow range (assuming the step-size is equal to one).

IV. Mechanism

We quantize and dequantize value in the memory hierarchy and keep the ISA, pipeline, load-store module, controller, and data-path intact for three reasons. First, providing support for quantization in the processing unit calls for invasive modifications in cores. Second, previous studies show that employing short variables in the computation part of the systems, such as ALU, can significantly increase the error (due to arithmetic overflow and inaccurate intermediate values) [9], [45]. Third, the cost of moving data is 6400 times higher than ALU operations in modern processors [7]. Therefore, we focused on reducing the cost of data movement. The location of conversion can be decided based on different tradeoffs (performance vs. power). Hereafter, we assume that the conversion occurs between L1 and L2 caches, as it provides higher speedup by increasing the effective size of L2. However, we also evaluate the performance improvement when the conversion point is between L2 and L3 cache such that values are stored in the full-length format in both L2 and L1 caches (Section VI-F). In this section we answer seven questions: (i) how to determine the quantization parameters?, (ii) how to transfer metadata to Hardware?, (iii) how to retrieve metadata?, (iv) how to locate quantized values?, (v) how to quantize and de-quantized values?, (vi) how to handle corner cases?, and (vii) how to avoid overflows?.

A. How to Determine the Quantization Parameters (Metadata)?

We observe that quantization parameters only depend on the nature of data and do not change significantly with different data sets. For example, many speech recognition applications process the amplitude of people's voice [22], [23], which does not drastically change across different datasets. In the modern development process, applications (such as machine learning applications) are trained and tested using some data set before the deployment. During the execution (inference) phase, the application process data with the same nature. Therefore, the quantization parameters can be derived using an offline profiler during the training and testing phase. To determine the effectiveness of the offline profiling, we divide our dataset into training and testing sets. Similar to machine learning training and testing, we kept at least 10% of the data for testing. For datasets such as NYSE [21], we tracked the stock prices for 20 days to have more than 80000 observations for training and ten days for testing. Some datasets such as CIFAR-10 [26] already have separated testing and training sets (There are 50000 training images and 10000 test images). We used their partitioning for testing and training. We profile applications using the training set to find the parameters for some specific average relative error (e.g., less than 10%/5%/1%) and find that the parameters provide similar accuracy even with the testing sets (the average relative error is 7%/2.2%/ 0.64%). Note that the quantization parameters can be selected conservatively with a small overhead. For example, a conservative selection of $0.5\times$*step-size* (the smaller the *step-size*, the lower the error) increases the bit-width only by *one bit*. Additionally, the offline profiling does not need to be 100% accurate as our mechanism is capable of handling a small fraction of outliers (explained in section IV-G).

Profiling can be a automated process. A parser detects arrays and the structure of arrays, then passes this information to the profiler. The profiler detects whether the arrays are large enough and extracts the quantization parameters for the specified tolerable error. A similar automated approach is being used for accelerator designs [10], [16].

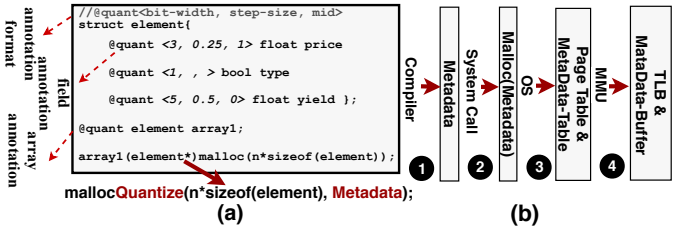


Fig. 8: Software-hardware interaction

B. How to Transfer Metadata to Hardware?

After profiling, we need to communicate the quantization parameters obtained by profiler to software and then from software to hardware. We have two options for transferring quantization parameters to software: (i) automatic and static annotation of arrays with quantization parameters as shown in Figure 8 (a), and (ii) putting the parameters in input arguments, so that it can be read at run-time, dynamically (useful for API and library developers).

There are three essential steps involved in the communication of quantization parameters to hardware. First, the compiler extracts the metadata from the annotated code. It extracts *Bit-width*, *Step-size*, and *Mid* values of each filed of the structure (or address of the variables, in which these values are stored), as shown in Figure 8 (a) and Figure 8 (b), step ❶. (For this step, we are mimicking the compiler support and do not change the compiler itself). Then, the compiler passes this data to OS, using a specialized malloc function (system call in step ❷). Second, the malloc function assigns a page aligned space to the array and calculates other required metadata (explained in Section IV-C). At this point, OS stores a few fields of metadata required in the critical path in the page table (we extended page table entries to store these fields), and the rest of metadata, in our proposed table, `MetaData-Table` (step ❸). Third, once a page is requested, memory management unit (MMU), as a part of its normal process, transfers page table entries to TLB, meaning that metadata stored in the page table are transferred to TLB automatically. Our customized MMU also transfers the corresponding metadata, stored in `MetaData-Table`, to our hardware module, called `MetaData-Buffer` (step ❹).

C. How to Retrieve Metadata?

Our proposed method requires two types of metadata. The first type of metadata, such as *DeQWordCount*, is required for address translation between L1 and L2. *DeQWordCount* determines how many words (of L1 values) can fit in a quantized L2 block. *DeQWordCount* is an integer number (in Section IV-F we explain why it should be an integer number). Access to *DeQWordCount* is in the critical path as it is required for sending a miss request to lower level cache (L2). Hence, *DeQWordCount* is stored in the TLB so that it can be accessed when the processor accesses the TLB for the traditional virtual to physical address translation. Existing systems, such as Intel x86-64 systems [46] have up to 15 unused bits in their TLB entries. Our proposed method requires 18 bits [47]. Accordingly, we only add three bits to the original TLB entry and the total overhead per core is 216 bytes (in a system with 64-entry first level TLB and 512-entry second level TLB). The second type of metadata, including *Step-size*, *Bit-width*, and *Mid*, is required for data conversion, We introduce a new hardware module, the `MetaData-Buffer`, to store the metadata required for data conversion. Thanks to our `MetaData-Buffer`, our method can mix compressible and non-compressible data. In our `MetaData-Buffer` (a full description of each field and its purpose is available at our online documentation [47]), for each variable of the structure, we have a one-bit field, called "Conv?" that determines whether that field needs conversion or not.

D. How to Locate Quantized Values?

When compression/decompression happens between two level of caches, page number and page offset of the values in the decompressed cache are different from those of compressed cache. Unlike most cache compression techniques, we do not need decoupling the tag array and the data array for address translation. In fact, our fixed bitwidth enables address translation using arithmetic operations, which is performed by two modules: (i) `BitLocationFinder`, and (ii) `Offset-Divider`.

We built upon prior works [29], [48] that allocate smaller page within a 4KB page and add few bits in the page table entries that point to the start of the smaller page within 4KB pages. Therefore we only need to translate page offset. We explain this process using two examples.

Example 1, an array of float numbers (float Arr[1000]: In this example, each element can be quantized with 5 bits, and each cache line of the system has 64 bytes (512 bits). As a result, each L2 block accommodates 102 quantized words(*DeQWordCount*=512/5=102 (the division is implemented by multiplication by reciprocal and takes 2.5 cycles)). Step❶, an L1 eviction happens. Step❷: `Offset-Divider` simply divides the word number of the virtual offset(Vofset) by *DeQWordCount*, to determine the index of the L2 block that contains the missed L1 block($index = (Vofset >> 2)/102$). Since the index is ready, L1 can send the write-back request to L2. Step❸: while L2 is finding the L2 block containing the evicted block, the `BitLocationFinder` finds the exact location of the evicted L1 block within the L2 block. To this end, the `BitLocationFinder` multiplies the quantized bitwidth by the remainder of the last division ($(Vofsett >> 2\%102) * 5$).

Example 2, an AoS: In this case, `BitLocationFinder` needs two metadata: (i) *AccLen* and (ii) *StartWord*. *AccLen* is a field of our `MetaData-Buffer`. Per each variable of the structure, *AccLen* stores the accumulated quantized length of preceding variables in the structure. This field optimizes `BitLocationFinder` to lower the complexity of address location calculation. *StartWord* is a field in the TLB extension that indicates with which word of the structure the page starts. `BitLocationFinder` first finds how many complete structures exist before the start of the L1 block and then multiply this number by the quantized size of the structure.

```
lengthOfCompleteStructures=(((Vofsett>>2)%
    DeQWordCount)/NumWordInStructure)*
    QntStructBits.
```

Then it uses *AccLen* and *StartWord* to calculate how many bits are required for any partial structure before the start of the L1 block. The remainder of the last division determines how many words exist in the partial structure.

```
remainder=(((Vofsett>>2)%DeQWordCount)%
    NumWordInStructure)
```

BitLocationFinder determines with which word of the structure, the partial structure ends

```
end=((Vofsett>>2)+StartWord)%
    NumWordInStructure
```

Then it calculates the number of bits in the partial structure and adds this to the lengthOfCompleteStructures.

```
if(end> remainder) {
    start=end-remainder
  BitLocationStart= AccLen[end]-AccLen[start
     ]+lengthOfCompleteStructures;
}else{
start=NumWordInStructure+end-remainder
BitLocationStart= AccLen[NumWordInStructure
    +1]-AccLen[start]+AccLen[end]+
    lengthOfCompleteStructures;
}
```

This operation, on cache misses, overlaps with reading the block from L2 and, on writebacks, lies out of the critical path and does not hurt performance (The above pseudo code is developed to simplify the explanation and differs from our optimized Verilog implementation).

E. How to Quantize and De-quantized Values?

The de-quantization process is on the critical path of a L1 miss whereas quantization only happens during the writebacks. Therefore, we design a parallel de-quantizer for efficient and fast conversion that performs the computation shown in Equation 1. Since the *Step-size* is a power of two number, the multiplication in Equation 1 can be implemented as a summation of the exponent of *Step-size* and exponent of $Index + QuantizedMid$ (In fact, we store the exponent of *Step-size* in the MetaData-Table) and a *leading one detector module* (LOD) can calculate the exponent of $Index + QuantizedMid$. We propose a 16-way parallel module where each of the ways takes a 32-bit word (extracted through a crossbar) from the cache block as its input and converts them in parallel (Figure 9). We synthesize the modules using an industry-standard 1xFinFET, and the latency of the whole process is 1.6 ns (4.8 cycles in 3GHz frequency).

$$\tilde{X} = (Index + QuantizedMid) \times step - size \quad (1)$$

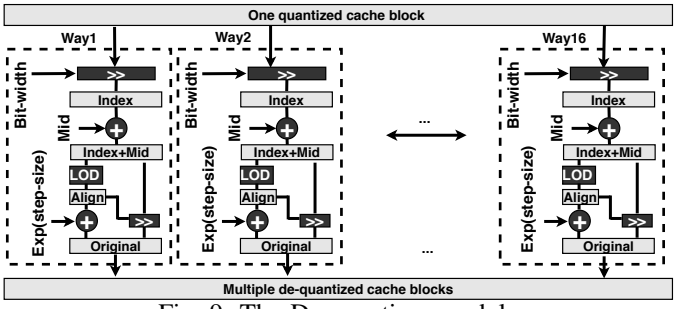


Fig. 9: The De-quantizer module

F. How to Handle Corner Cases?

As an extra optimization, we relax the constraint of fitting only integer number of L1 blocks in L2 blocks to avoid the overhead of padding. As a result, naively unpacking a L2 block generates one or two partial L1 blocks on de-quantization, wasting cache space and requiring some extra metadata per cache block to track the valid words of the block. Alternatively, we place the partial blocks in the prefetch buffer until the rest of the block arrives to avoid extra metadata and wasted space in L1. We evaluate a 4-entry prefetch buffer (similar to the original prefetch-buffer paper [49]). As our method is word-aligned, we only add one valid bit per word to the prefetch buffer (total overhead of 64 bits). On a L1 miss, our Offset-Divider generates a miss for the L2 cache block that contains the missed address. Subsequent requests to the valid part of the partial block are serviced from the prefetch buffer and cause no miss. Once an address from the rest of the partial block is requested, the Offset-Divider module will generate a miss request for the subsequent L2 block. Note that we always store an integer number of quantized words in an L2 block (determined by *DeQWordCount*). Therefore, No L1 word spans two L2 blocks, ensuring that Offset-Divider never generates two L2 miss for one word, avoiding complexity in miss status history register (MSHR).

G. How to Avoid Overflows?

We provide two options for dealing with the outliers that does not fit in short bitwidth and may cause overflows: *(i)* mapping the outliers to the maximum or minimum of the range that can be represented using the *same* bit-width used in the quantization process, and *(ii)* storing the outliers in the original floating point format in a separate space. We explain how we support the second option using a walk-through example that shows a scenario with an outlier in a block.

Step ❶: L1 evicts a cache block. Step ❷: the converter tries to quantize the evicted block and send it to the L2 but it detects an oulier in the evicted block. Step ❸: the converter fills the location of the value in the L2 block with an outlier indicator (we assign both all-one or positive-all-one values as outlier indicators. For example, for a quantized value with bit-width of four, outliers will be stored as "1111"). Therefore, the layout of data in L2 does not change. Step ❹: to preserve the real value of the outlier, the converter also sends a message to memory, containing the real value of the outlier along with Voffset of the value to be stored in the reserved space, extra b blocks at the end of each page, after the quantized values (In case the reserved block space cannot accommodate a new outlier, the memory controller informs the OS to convert the quantized page to a normal page). Step ❺: when this block is requested again (on a L1 miss), the converter sees the indicator and detects that the related value is stored as an outlier. Thus, the converter sends a memory request to get the real value of the outlier from the reserved space. To this end, the memory controller reads the reserved blocks for outliers and scans these blocks until it finds the Voffset of the requested variable. The real value of the outlier is stored next to the Voffset. Therefore, while L2 stores the outlier indicator, with the same bitwidth of quantized values, L1 always stores the real value of the outlier

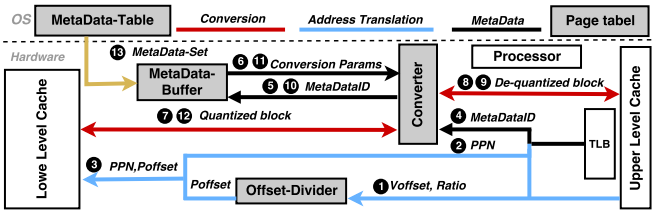


Fig. 10: Overall architecture of hardware quantization

and we never give up the wide range that floating point format can support.

In our evaluation with real data sets, we notice that the fraction of outliers is significantly low (less than 0.005%) and the overhead of retrieving the outliers does not hurt the performance (evaluated in Section VI-F).

V. More Walk-Through Examples

In this section, we explain the necessary steps during loading a new page, a L1 miss, and a L1 eviction. The steps are marked in Figure 10.

Steps during loading a new page: There are three steps involved in this process: (i) once processor loads a new page, our proposed boolean filed, *IsQuant* in the TLB entry indicates whether the page is quantized or not, (ii) if the page is quantized, our proposed *MetaDataID* field of the TLB entry specifies the Id of the corresponding `MetaData-Set` in `MetaData-Table` of the corresponding virtual space, (iii) when a TLB entry is loaded, the Address Space ID (ASID) + *MetaDataID* of the TLB entry is compared against the ASID+ *MetaDataID* of all the sets. In case of a miss, MMU assigns one of the free sets in the `MetaData-Buffer` to the *MetaDataID* and loads the corresponding `MetaData-Set` of the `MetaData-Table` in the `MetaData-Buffer` (Figure 10 ⓭). Note that each TLB entry traditionally has the ASID. Accordingly, we store ASID+ *MetaDataID* in each set of the `MetaData-Buffer`.

Steps during a L1 cache miss: There are three steps.

1. finding the address of the L2 block that contains the quantized values: Traditionally, for each load or store instruction, processor uses virtual offset (*Voffset*) to access L1 and virtual page number (*VPN*) to read the corresponding TLB entry and extract the the physical page number (*PPN*). In our method, on a L1 miss, if the *IsQuant* of the TLB entry indicates that the missed address belongs to a quantized page, L1 sends the *Voffset* to the `Offset-Divider` (Figure 10 ❶) and send *PPN* to L2 (❷). The translator divides the *Voffset* by the *DeQWordCount* value to generate the page offset (*Poffset*) of the L2 block and sends this offset to the L2 cache (❸).

2. Getting quantization parameters: While, waiting for the L2 block to be received, the TLB sends the *MetaDataID* to the `Converter` (❹). The `Converter` uses the *MetaDataID* to read the quantization parameters from the `MetaData-Buffer` (❺ and ❻).

3. Conversion: When L2 sends the quantized block to the `Converter` (❼), the `Converter` uses the quantization parameter to de-quantize the block and sends back multiple de-quantized blocks to L1 (❽).

Steps during a L1 write-back: There is no TLB access during the write-back. Accordingly, we store the *MetaDataID*, in the L1 cache tag. Quantization in write-back has two steps.

1. Getting quantization parameters: When L1 evicts a dirty block from a quantized page (❾), the *MetaDataID* is read from its tag and the `Converter` uses this *MetaDataID* value to read the quantization parameters from the `MetaData-Buffer` (❿ and ⓫).

2. Conversion: The `Converter` quantizes the block and calculates the location of the block in the quantized L2 block (using the `BitLocationFinder` module), and sends the quantized block to L2 (⓬). Upon receiving the block, L2 updates data at the correct location within the quantized block.

TABLE II: The configuration of simulated systems

Component	Parameters
Processor	4/8/16 cores, 3 GHz, 8-wide issue
L1	64B cache-line, 2-way associative, 32KB, private, 4 cycles
L2	8-way associative, 256KB, private, 12 cycles
L3	16-way associative, 8/16/32 MB, shared, 36/43/58 cycles
Prefetch buffer	4-entry
Memory system	DDR3_1600_x64, 2 channels

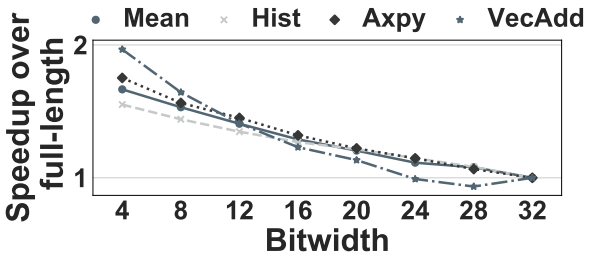
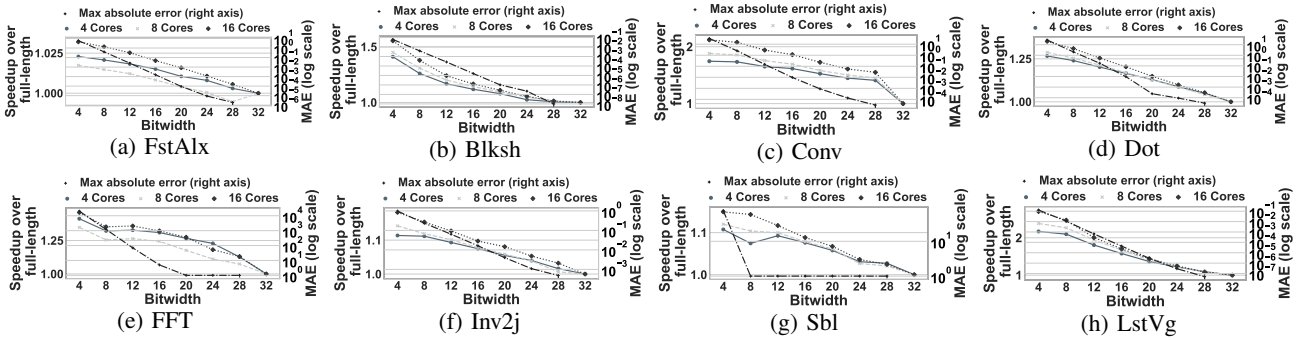


Fig. 11: Effect of bitwidth on speedup for microkernels.

VI. Evaluation

A. Methodology

To evaluate the performance and energy consumption of our proposed method, we augmented Gem5 [50] and McPAT [51] with timing and energy model of our proposed components. For simulation configuration, we use the specification of Intel Core-i7 Haswell processor (listed in Table II) and add a penalty of five cycles (explained in Section IV-E) to model the conversion latency of a parallel converter module. We evaluated four microkernels: (i) Hist (calculates the histogram of a vector of values), (ii) Mean (calculates the mean of a vector), (iii) Axpy ($Y = \alpha \times X + Y$), and (iv) VecAdd ($Z = X + Y$). We also evaluated eight applications from different domains: four applications from AxBench [27] (Blksh, Sbl, Inv2j, and FFT), three convolution with configuration of three different layers of the deep learning application with high, moderate, and low locality (FstAlx, Conv, and LstVg), and a Dot product kernel (Dot). For the eight applications, we used standard data sets for each application [21], [22], [23], [24], [25], [26], [52]. Our evaluated workloads represent modern



(a) FstAlx (b) Blksh (c) Conv (d) Dot

(e) FFT (f) Inv2j (g) Sbl (h) LstVg

Fig. 12: Effect of bitwidth on speedup and maximum absolute error

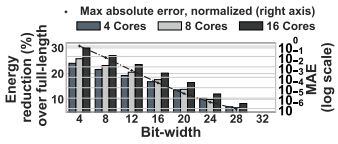


Fig. 13: Energy reduction

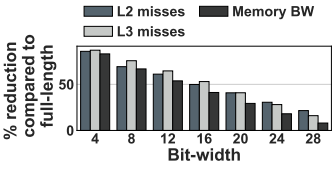
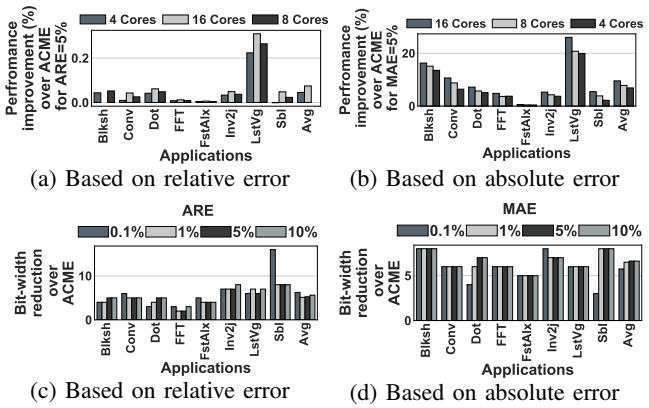


Fig. 14: Source of improvement



(a) Based on relative error (b) Based on absolute error

(c) Based on relative error (d) Based on absolute error

Fig. 15: Comparison against ACME [9]

applications, which are highly memory-intensive and exhibits a large memory footprint. Prior works have shown that these applications are bottlenecked by the cost of data movement [3], [7], [53]. However, in order to be comprehensive, we also evaluated two computation-intensive applications (FstAlx and Inv2j), which inherently do not benefit from reducing the cost of data movement.

B. Performance Improvement and Error Analysis

Figure 11 illustrates the effect of bitwidth on speedup for four microkernels. Mean has one memory access per compouation, Hist has two memory accesses but one of them is often cached, Axpy has two memory accesses per computation and VecAdd has three memory accesses per computation. This figure shows that the narrower the bitwidth, the higher the speedup. It also shows that for VecAdd with three memory access per computation, a small reduction in bitwidth (e.g. bitwidth of 24) can not compensate the conversion penalty. However, for smaller bitwidth, such as four, the reduced memory bandwidth provide a significant speedup. Figure 12 shows the speedup (on the left axis) and maximum absolute error (on the right axis) per bitwidth for eight applications using real data sets. Our proposed mechanism achieves on average 1.6× (up to 2.7×) speedup over the baseline system.

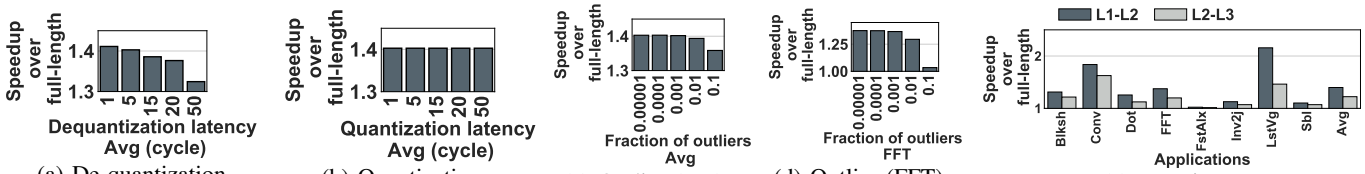
There are four observations that we draw from Figure 12. First, the shorter the *bit-width*, the higher the speedup, demonstrating that reduction in data movement directly impacts the overall performance. In rare cases, such as Sobel, access pattern with the 8-bit variables causes more conflict misses than 12-bit variable, causing unexpected slowdown in a 4-core system. Second, a system with a higher number of cores gains higher speedup from reduction in data movement, due to the higher contention in caches and memory. Third, applications with a low temporal locality, such as LstVg, gains the most benefit from our proposed method. On the contrary, applications that have a high temporal locality gain the least speedup (e.g., FstAlx [54] achieves less than 1.04× speedup). Fourth, the shorter the *bit-width*, the higher the maximum absolute error, except for applications such as Sbl, where the values (pixel values) inherently can be represented by 8 quantized bits, without significant accuracy loss.

C. Energy Consumption

We model the energy consumption of all memory elements (such as `MetaData-Table`, etc.) and all arithmetic operations using McPAT [51]. Figure 13 demonstrates the average energy reduction of the evaluated benchmarks over the baseline when we vary *bit-width* and number of cores. It shows that the energy cost of data movement decreases when the applications use shorter *bit-width*.

D. Source of Improvement

The performance and energy benefits of quantization stem from two sources: *(i)* reducing the number of cache misses (due to an increase in the effective size of L2 and L3 and also due to loading one quantized block instead of multiple full-length blocks), and *(ii)* decreasing the memory bandwidth consumption. Figure 14 demonstrates that reducing the variable length from 32 bits to 6 bits reduces L2 misses/L3 misses/BW consumption on average by 77%/82%/74% in a 16-core system. None of the evaluated application fit in L1



(a) De-quantization (b) Quantization (c) Outlier (avg) (d) Outlier (FFT) (e) Location

Fig. 16: Sensitivity analysis

and L2. Since the L3 is shared among cores, LstVg does not fit even in its share of L3 (before quantization). Some applications such as Blksh, fit in L3, but they do not show locality with long reuse distance, so they experience a high L3 miss rate. As an example, for Blksh, the L1 miss rate is 0.006052, the L2 miss rate is 0.877949, and the L3 miss rate is 0.999938. For FFT, L1 miss rate is 0.039064, L2 miss rate is 0.966428, and L3 miss rate is 0.34570956478. for LstVg, the L1 miss rate is 0.006861, the L2 miss rate is 0.917308 and 0.9999.

E. Comparison with Prior Works

Figure 15 compares the performance of our proposed method against ACME [9] that omits the LSBs to shorten the variable lengths. We evaluate both of the mechanisms with the specific *bit-width* that achieves the same level of average relative error (ARE) and maximum absolute error (MAE). Figure 15 (a) and 15 (b) show that our proposed method can achieve up to 31%/26% performance improvement with ARE/MAE fixed to 5%. We provide the same accuracy with shorter variables and therefore, achieve higher speedup. Figure 15 (c) and 15 (d) show the reduction in the number of bits compared to ACME when we vary the accuracy. On average, the bit width reduction in the quantized variables are 5.62/5.25/5.1/6.2 bits (ARE) and 6.62/6.62/6.5/5.7 bits (MAE) when the error rate is 10%/5%/1%/0.1%.

F. Sensitivity Analysis

In this section, we analyze the effect of four parameters: (i) de-quantization latency, (ii) quantization latency, (iii) the fraction of the outliers, and (iv) the location of the conversion in the memory hierarchy, in a system with 4 cores.

(i) De-quantization and (ii) Quantization latency. The conversion latency depends on many design parameters (e.g. technology size, parallel vs serial `Converter`, etc.). Figure 16(a) shows that for a de-quantization latency as large as 20 cycles, our proposed method achieves 1.38× speedup on average. The low sensitivity to the quantization latency stems from the fact that a quantized L2 block fills multiple de-quantized L1 blocks and consequently subsequent L1 hits amortize the additional conversion penalty on a L1 miss. The speedup is *insensitive* to the quantization latency (Figure 16(b)) because quantization occurs on writeback requests which are out of critical path.

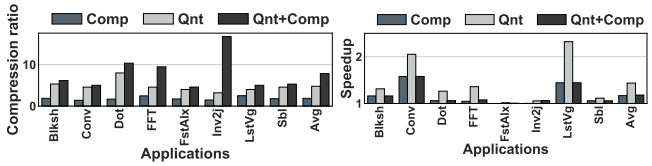
(iii) The fraction of the outliers. We find that the fraction of the outliers is very low (the highest rate belongs to Blackscholes and FFT where the fraction is 0.00001, 0.00002, respectively) that they impose a negligible overhead. In order to analyze the overhead of the outliers, we synthetically varied the fraction of outliers from 0.00001 to 0.1. Figure 16(c) shows that the fraction of outliers does not affect the performance except for a high fraction such as 0.1. The *insensitivity* to the fraction of outliers stems from the fact that the access to outliers is not on the critical path unless the outlier is in the missed L1 block (and not in the rest of unpacked blocks). Figure 16(d) shows that for some applications such as FFT (with relatively high L2 miss rate), the accesses to outliers may occur on the critical path and in that case, even a lower fraction such as 0.01 can significantly impact the performance.

(iv) Location of the conversion. So far, we evaluated the performance of our proposed method when data conversion occurs between L1 and L2. Figure 16(e) evaluates the alternative design, where conversion happens between L2 and L3. As a result, the conversion happens less frequently, decreasing the energy consumption. This figure shows that placing the conversion point between L2 and L3 provides on average, 1.22× and up to 1.4× speedup.

G. Quantization vs. Compression

The most related cache compression technique that takes into account the effect of data type and the effect of floating point is Hycomp [17].

Figure 17(a) illustrates that Hycomp yields only a compression ratio of 1.85 on average whereas our method itself yields the ratio of 4.78. Although, quantization on top of compression achieves a higher compression rate (7.83), a variable cache block size of compresssion will again require a complex mechanism for finding the location of cache blocks. We evaluated the effect of decoupling the tag array and data array in the last level cache. Figure17 (b) shows that the latency of access to the decoupled tag and data array nullifies the higher compression rate that can be achieved by employing compression on top of quantization. Figure 18 demonstrates



(a) Compression ratio (b) Speedup

Fig. 17: Quantization vs. compression

that address translation by `BitLocationFinder` and corner case handling (Section IV-F) outperforms padding (which is proposed by LCP [29] to avoid the cost of decoupling tag

array and data array), on average by 20% and up to 45%, in a 4-core system, where average bit-width is 18.

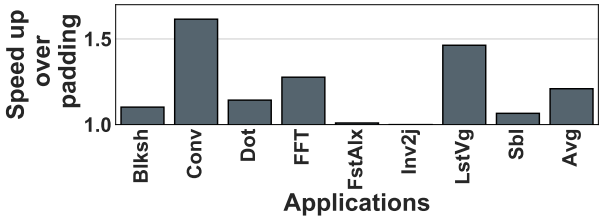


Fig. 18: Padding (LCP) vs. BitLocationFinder

H. Software-based Quantization vs Hardware-based Quantization

Figure 19 demonstrates that our hardware-based design delivers 1.54× speedup compared to software-based approach, in a 4-core system. The speedup stems from less data movement and lower conversion overhead. We tried our best to manually optimize the code and perform conversion only when it is required. One of the benefits of our method is that the programmer does not need to think about optimizations, overflows, and outliers for every line of arithmetic computations in the code. In software, for applications such as Blackscholes we had to use standard 32-bit variables to cover the outliers that require 18 bits. With quantization in hardware we can employ 6-bit variables for Blacksholes and store outliers in their original format in a separate space.

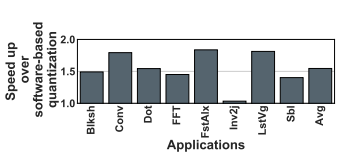


Fig. 19: Quantization in hardware vs. software

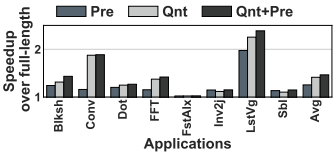


Fig. 20: Quantization vs. prefetching

I. Quantization vs. Prefetching

Unlike our method, prefetching can not alleviate the energy cost of data movement and do not increase the effective size of caches or decrease the memory bandwidth consumption. However, we can employ quantization in tandem with a prefetcher to hide the conversion latency. Prefetcher tracks physical addresses and does not need address translation. Therefore, we modified the prefetcher to record the *MetaDataID* for the previous misses and use it for de-quantizing the prefetched blocks. Figure 20 shows that our method achieves on average 1.6× speedup over prefetcher. Figure 20 also demonstrates that our method along with prefetchers can achieve on average 1.46× speedup over the baseline system.

J. Hardware Overhead

We design and synthesize the modules using an industry-standard 1xFinFET technology with foundry models.The total area of our modules, per core, is 0.012455 mm^2. Accordingly, 16 modules in a 16-core processor impose 0.2% area overhead (we scaled the area of the processor in 22 nm technology, obtained from McPAT [51], to derive the area of the processor in 1xFinFET technology). The total overhead is 0.25% of the original processor.

VII. Related Work

Bit-width Reduction Techniques. Several studies employed the OLSB method for ASIC and FPGA designs to customize the *bit-width* [10], [14]. Prior works also show the energy reduction of quantization with arbitrary *bit-width* in accelerator design and provide a framework for deciding the *bit-width* [15], [16]. However, these solutions cannot be adopted in the general-purpose processors. The only general-purpose solution, ACME [9] employs the OLSB method which has lower accuracy and lower performance than quantization.

Software-based Quantization for Specific Applications. Due to high overhead, software-based quantization is not a popular technique unless in three following cases: (i) to compress data before storing in storage [55], (ii) to compress data before offloading data to accelerators [7], [56], and (ii) in some cheap embedded processors [57], [58] that cannot accommodate floating point ALU, where the cost of conversions is less than the emulation of floating point operations on integer ALU and programmer's effort for manipulating integers pays off.

Approximate cache techniques. Cache approximation techniques [59], [60] also reduce the cost of movement and storage of data. We do not compare against any other approximate cache techniques, as ACME [9] (to which we have compared our method in Section VI-E) shows performance improvement over the prior approximate cache techniques, Doppelganger [59].

VIII. Conclusion

This is the *first* work to argue that *quantization* can be adopted as a technique to accelerate memory-intensive applications in general-purpose processors. We evaluate 11 *real data sets* and demonstrate that many applications operate on a limited range of values, making quantization widely applicable and effective in shortening the *bit-width*. To this end, we proposes a software-hardware interface, whereby we transfer information of our high-level abstraction to hardware modules and transparently convert the short variables to the original format before being used in computation. Our evaluation demonstrates that quantization provides significant performance and energy improvement over the state-of-the-art bit-reduction techniques. We believe that our architectural support, popularize quantization as a low-overhead, high-accuracy approximation technique in general-purpose processors.

IX. Acknowledgment

We would like to thank the anonymous reviewers, and Kevin Skadron for their valuable feedback and suggestions.
This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA.

REFERENCES

[1] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 56–65.

[2] D. Pandiyan and C.-J. Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 171–180.

[3] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 316–331.

[4] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "A scalable and efficient in-memory interconnect architecture for automata processing," *IEEE Computer Architecture Letters*, 2019.

[5] I. Akturk and U. R. Karpuzcu, "AMNESIAC: Amnesic automatic computer," in *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[6] M. Lenjani and M. R. Hashemi, "A novel arbitration scheme for bandwidth and jitter guarantees in asynchronous nocs," in *2009 14th International CSI Computer Conference*. IEEE, 2009, pp. 231–235.

[7] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.

[8] M. Lenjani and M. R. Hashemi, "Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities," *IET Computers & Digital Techniques*, vol. 8, no. 1, pp. 30–48, 2014.

[9] A. Jain, P. Hill, S.-C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[10] A. Lotfi, A. Rahimi, H. Esmaeilzadeh, and R. K. Gupta, "SqueezCL: Squeezing OpenCL kernels for approximate computing on contemporary GPUs," in *Workshop on Approximate Computing*, 2015.

[11] M. Henneaux and C. Teitelboim, *Quantization of gauge systems*. Princeton university press, 1994.

[12] R. M. Gray, "Vector quantization," in *Readings in speech recognition*. Elsevier, 1990, pp. 75–100.

[13] A. Gersho and R. M. Gray, *Vector quantization and signal compression*. Springer Science & Business Media, 2012, vol. 159.

[14] S. Lee, D. Lee, K. Han, E. Shriver, L. K. John, and A. Gerstlauer, "Statistical quality modeling of approximate hardware," in *Quality Electronic Design (ISQED), 2016 17th International Symposium on*. IEEE, 2016, pp. 163–168.

[15] T. Moreau, F. Augusto, P. Howe, A. Alaghi, and L. Ceze, "Exploiting quality-energy tradeoffs with arbitrary quantization: special session paper." ACM, 2017, p. 30.

[16] T. Moreau, F. Augusto, P. Howe, A. Alaghi, and L. Ceze, "QAPPA: A framework for navigating quality-energy tradeoffs with arbitrary quantization," *Technical Report UW-CSE-17-03-02*, 2017.

[17] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 38–49.

[18] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 62–73.

[19] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 377–388.

[20] "Verilog implementation of quantizer and de-quantizer modules ," https://github.com/quantizerdequantizer/QuantizerDeQuantizer.

[21] "NYSE Stock Exchange," http://online.wsj.com/mdc/public/page/2_3024-NYSE.html.

[22] N. Hammami and M. Bedda, "Improved tree model for arabic speech recognition," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 5. IEEE, 2010, pp. 521–526.

[23] "Spoken Arabic Digit Data Set ," https://archive.ics.uci.edu/ml/datasets/Spoken+Arabic+Digit.

[24] "EEG Eye State Data Set," https://archive.ics.uci.edu/ml/datasets/EEG+Eye+State.

[25] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[26] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html.

[27] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh, "AXBENCH: A multi-platform benchmark suite for approximate computing," *IEEE Design & Test*, 2016.

[28] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[29] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 172–184.

[30] "Credit Card Fraud Detection ," https://www.kaggle.com/dalpozz/creditcardfraud.

[31] "Air Quality Data Set ," https://archive.ics.uci.edu/ml/datasets/Air+quality.

[32] "Forest Fires Dataset ," http://www3.dsi.uminho.pt/pcortez/forestfires/.

[33] "Parkinson Speech Dataset with Multiple Types of Sound Recordings Data Set ," https://archive.ics.uci.edu/ml/datasets/Parkinson+Speech+Dataset+with++Multiple+Types+of+Sound+Recordings.

[34] "Kdd Cup 99 dataset ," http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[35] "Energy Efficiency dataset ," http://archive.ics.uci.edu/ml/datasets/energy+efficiency.

[36] "Sales Transactions Dataset Weekly Data Set ," https://archive.ics.uci.edu/ml/datasets/Sales_Transactions_Dataset_Weekly.

[37] J. W. Osborne and A. Overbay, "The power of outliers (and why researchers should always check for them)," *Practical assessment, research & evaluation*, vol. 9, no. 6, pp. 1–12, 2004.

[38] V. Barnett, T. Lewis *et al.*, *Outliers in statistical data*. Wiley New York, 1994, vol. 3, no. 1.

[39] J. Van den Broeck, S. A. Cunningham, R. Eeckels, and K. Herbst, "Data cleaning: detecting, diagnosing, and editing data abnormalities," *PLoS medicine*, vol. 2, no. 10, p. e267, 2005.

[40] H. Liu, S. Shah, and W. Jiang, "On-line outlier detection and data cleaning," *Computers & chemical engineering*, vol. 28, no. 9, pp. 1635–1647, 2004.

[41] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 111–121.

[42] C.-K. Luk and T. C. Mowry, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 134–141, 1999.

[43] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 25–32.

[44] J. Kepner, D. Bade, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[45] A. Jain, P. Hill, M. A. Laurenzano, M. E. Haque, M. Khan, S. Mahlke, L. Tang, and J. Mars, "Cpsa: Compute precisely store approximately," in *Workshop on Approximate Computing Across the Stack*, 2016.

[46] "Intel 64 and IA-32 Architectures Developerś Manual ," https://https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[47] "Full description of metadata ," https://github.com/quantizerdequantizer/quantizerdequantizer/blob/master/Full_description_of_metadata/.

[48] E. D. Berger, "Memory management for high-performance applications," Ph.D. dissertation, University of Texas at Austin, 2002.

[49] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE, 1990, pp. 364–373.

[50] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The Gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[51] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.

[52] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[53] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 639–651.

[54] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[55] J. E. Fowler, "Qccpack: An open-source software library for quantization, compression, and coding," in *Applications of digital image processing xxiii*, vol. 4115. International Society for Optics and Photonics, 2000, pp. 294–302.

[56] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 13–24.

[57] "Comparing Fixed- and Floating-Point DSPs ," http://www.ti.com/lit/wp/spry061/spry061.pdf.

[58] "Fixed-Point Digital Signal Processors ," http://www.ti.com/lit/ds/symlink/tms320c6413.pdf.

[59] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: a cache for approximate computing," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 50–61.

[60] J. San Miguel, J. Albericio, N. E. Jerger, and A. Jaleel, "The bunker cache for spatio-value approximation," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.