Approximate Data Reuse-based Accelerator Design for Embedded Processor

HISASHI OSAWA and YUKO HARA-AZUMI, School of Engineering, Tokyo Institute of Technology, Japan

Due to increasing diversity and complexity of applications in embedded systems, accelerator designs trading-off area/energy-efficiency and design-productivity are becoming a further crucial issue. Targeting applications in the category of Recognition, Mining, and Synthesis (RMS), this study proposes a novel accelerator design to achieve a good trade-off in efficiency and design-productivity (or reusability) by introducing a new computing paradigm called “approximate computing” (AC). Leveraging from the facts that frequently executed parts of applications (i.e., hotspots) are conventionally the target of acceleration and that RMS applications are error-tolerant and often take similar input data repeatedly, our proposed accelerator *reuses previous computational results of similar enough data* to reduce computations. The proposed accelerator is composed of a simple controller and a dedicated memory to store limited sets of previous input data with corresponding computational results in a hotspot. Therefore, this accelerator can be applied to different and/or multiple hotspots/applications only through small extension of the controller, to achieve efficient accelerator design and resolve the design-productivity issue. We conducted quantitative evaluations using a representative RMS application (image compression) to demonstrate the effectiveness of our method over conventional ones with precise computing. Moreover, we provide important findings on parameter exploration for our accelerator design, offering a wider applicability of our accelerator to other applications.

CCS Concepts: • **Computer systems organization** → **Embedded hardware**;

Additional Key Words and Phrases: Embedded processor, approximate computing, data reuse

ACM Reference format:

Hisashi Osawa and Yuko Hara-Azumi. 2019. Approximate Data Reuse-based Accelerator Design for Embedded Processor. *ACM Trans. Des. Autom. Electron. Syst.* 24, 5, Article 56 (August 2019), 25 pages.
<https://doi.org/10.1145/3342098>

1 INTRODUCTION

An ever-increasing diversity and complexity of embedded systems applications, particularly in the Internet-of-Things (IoT) era, demand the customization of embedded processors for each application to achieve efficient speedup and energy reduction while increasing design-productivity. Conventionally, hotspots (i.e., frequently executed and/or computationally heavy parts) of applications are first identified. Then, a general embedded processor is extended with accelerators tailored to

This work is partially supported by JSPS KAKENHI 17H04677. The authors acknowledge the support from the VLSI Design and Education Center (VDEC), the University of Tokyo. The authors would also thank Fransiscus Marcel Satria for his supports in conducting experiments.

Authors' addresses: H. Osawa and Y. Hara-Azumi, School of Engineering, Tokyo Institute of Technology, Japan, S3-50, 2-12-1 Ookayama, Meguro, Tokyo, 152-8552; emails: {hisashi, hara}@cad.ict.e.titech.ac.jp.

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.

© 2019 Association for Computing Machinery.

1084-4309/2019/08-ART56 \$15.00

<https://doi.org/10.1145/3342098>

those hotspots. In general, the more critical the hotspots are, the larger the circuit area of their accelerators is. Hence, deployable accelerators are limited in most embedded systems because of their stringent design constraints [4, 9].

These years, a new computation paradigm called *approximate computing* (AC) has been highlighted in a variety of applications, which are categorized as Recognition, Mining, and Synthesis (RMS) [26]. This paradigm is particularly suitable in relaxing the computation complexity for area reduction, speedup, and/or energy reduction by accepting some error. Because of not only such admirable impacts on hardware design but also an increase in RMS applications, which are known to be error-tolerant [1, 29], a number of AC techniques for both hardware and software have been extensively studied to cope with the complexity of embedded system designs [8, 13, 15–17, 20–22, 25, 26, 28].

Existing AC techniques can be roughly classified into two types from the perspective of *granularity*: fine-grained techniques (at the operation-level) and coarse-grained techniques (at the task-level). The fine-grained techniques aim at relaxing every operation, mainly on hardware (the register-transfer or transistor levels) for the sake of critical path delay reduction, such as (segmented) computational resources (e.g., adder and multiplier) [8, 21, 27, 28] and least significant bits (LSBs) truncation [13, 20]. These techniques are well suited for relatively small, simple systems like DSP circuits [7, 13, 20]. Contrarily, the coarse-grained techniques aim at reducing the amount of computations, such as task skipping [17, 18], input sampling [22], pruning [25], and data reuse [6, 14, 15], and they are more suitable for relatively large, complex systems, like multicore processors with multiple memory hierarchies [5, 6, 14, 15, 17, 22, 25].

In this article, we address the conventional precise-computing hotspots limitations by employing an AC technique for accelerator's design. More specifically, we focus on “*data reuse*,” which has been mainly utilized to resolve cache miss penalties [15, 18], cache footprint reduction [14], or synchronization penalties [17] in multicore processors, but has been rarely utilized for computational reuse particularly in single-core processors. Among existing works in AC, References [5, 6, 23] proposed the most relevant approaches in computation-reuse architectures, but they presume that processors have rich memories and resources, which cannot be expected in embedded systems. Moreover, they have two issues in the flexibility and scalability; Because the reusable approximate data are statically pre-set in a lookup-table based on a profile, they cannot flexibly take into account the run-time changes of input data. Also, because they assume to target a hotspot with a single or a few approximatable outputs, it is not scalable in handling hotspots with several approximatable outputs or more.

We propose an architectural approach that resolves the above issues by *dynamically* updating the reusable data and handling hotspots *irrespective of the number of approximatable outputs*. Our approach has four essential features as follows: (1) *applicable to embedded systems* under stringent constraints on circuit area, (2) capable of efficiently speeding up target applications by reusing *similar enough previous data (computations)* in a hotspot, (3) capable of speeding up *multiple* hotspots using a *single* accelerator with a small extension of the accelerator's controller, and (4) easily *applicable to multiple hotspots in an application and/or in different applications* by varying some parameters. Thus, our approach is flexible and scalable in tackling with a variety of applications and their hotspots. To demonstrate the effectiveness of our study, we use a single-core embedded processor as a baseline architecture and accelerate multiple hotspots by using the proposed data reuse-based accelerator.

A preliminary version of this study appeared in Reference [16]. In this extended version, we elaborate two features (1) and (2) using a variety of input data. Additionally, we achieve feature (3) to enhance the flexibility and effectiveness of our study. For simplicity and comprehensibility, a case study is reported herein to explain an image compression algorithm, one of the representative

RMS applications, where two distinct hotspots are the targets of acceleration to evaluate feature (4). Through quantitative evaluations, we demonstrate the effectiveness of the proposed method over conventional methods, disclose important findings of the parameter exploration, and further discuss feature (4). By these evaluations, we reveal useful parameter settings to handle multiple hotspots, which is essential for the practical use.

In summary, this article has significant extensions from Reference [16] and research contributions over existing techniques as follows:

- Leveraging the features of RMS applications, we propose an approximate data reuse-based accelerator design composed of both hardware (architecture) and software (compilation) techniques. Our accelerator can efficiently accelerate the target hotspots and is thus applicable to embedded systems with stringent design constraints by adequately setting the design parameters.
- Our accelerator can accelerate multiple hotspots of a target application through a small extension of the accelerator's controller, unlike conventional (precise-computing) methods, which require a dedicated accelerator for each hotspot.
- Our accelerator has a parameterized structure that can be flexibly tuned depending on the reusable data and the degree of reusing approximate data. Such flexibility enables to accelerate multiple hotspots that have different characteristics (i.e., having different data to be reused and the different tolerance to errors caused by the approximate data reuse). In other words, this accelerator design achieves good applicability and design-productivity for multiple hotspots of a target application or even for different applications.
- In our evaluation, we exhaustively explore and examine useful parameter combinations of our accelerator through a case study using a whole, concrete image compression application where two hotspots with different characteristics are accelerated.

The remainder of this article is organized as follows: Section 2 describes a brief explanation of our target application and a motivational example to reveal limitations of conventional accelerator designs based on precise computing. Next, Section 3 provides the detailed descriptions of our proposed accelerator from the hardware and software viewpoints. Section 4 then demonstrates the effectiveness of the proposed method in terms of circuit area, performance and energy consumption over conventional methods, discussing the parameter exploration. Finally, Section 5 concludes this article.

2 PRELIMINARY

This section first briefly reviews an image compression algorithm utilized for evaluations and then describes our motivational example.

2.1 Image Compression

Media processing applications are the representatives of RMS applications [26]. Because many of these applications have been frequently deployed on embedded systems, their accelerator designs have been intensively studied for decades. Image compression, e.g., lossy compression of a large raw image to a JPEG image (hereafter referred to as "JPEG compression"), is one of the media processing applications. Its importance in embedded systems will further increase among a number of RMS applications because of the growing number of image sensing devices and the need for compressing data under a limited communication bandwidth in the IoT era.

In this article, we evaluate our accelerator on the JPEG compression application. Because this application has complex features that can be found in practical applications, i.e., having multiple hotspots with many approximatable outputs rather not but a few (the details will be provided

Fig. 1. JPEG compression.

in Section 3.4), our evaluations will show the applicability and transferability of our accelerator to general RMS applications. Also, exploring and demonstrating how our approximate data reuse approach can exploit the error-producing features of the lossy compression applications will bring new, interesting research opportunities in approximate computing.

Figure 1 explains the algorithmic flow of the JPEG compression performed mainly in the following nine steps, each of which is described as a rectangle in the figure:

- **Read File:** Reads pixels from an input raw image (.img)
- **Read 16 Lines:** Extracts 16-line pixels from the input image. The following steps are all performed by the 16 lines
- **RGB-to-YCbCr:** Converts each pixel of the input image from RGB (expressed with three primary colors: red, yellow and blue) to YCbCr (luminance (Y) and two types of chrominance (Cb and Cr))
- **Downsampling:** Compresses the data, particularly in terms of two chrominance elements
- **8 × 8 Blocking:** Constructs an 8 × 8 block window for each channel of Y, Cb, and Cr. The remaining processes are performed on the 8 × 8 block in the order of Y, Y,¹ Cb, and Cr channels
- **Discrete Cosine Transform (DCT):** Resolves the frequency of each block
- **Quantization:** Compresses data to reduce the data size
- **Huffman Encoding:** Performs an entropy encoding to further compress the output image
- **Write File:** Writes the results in the JPEG format (.jpg)

If designers would like to accelerate a target application, then they need to analyze the computational breakdown at a procedural level (e.g., at the function level) and then identify the critical parts of the application (i.e., hotspots), because accelerating hotspots is the most effective. Figure 2 shows the profile of the computational breakdown of the JPEG compression algorithm. Obviously,

¹The Y channel is processed twice consecutively.

Fig. 2. Computational breakdown of JPEG compression.

(a) Computational breakdown of a target application

(b) Area overhead and performance improvement

(c) Energy saving and output accuracy

Fig. 3. A motivational example.

two parts (highlighted as red rectangles in Figure 1) dominate the computational time.² The most critical hotspot is obtained from 8×8 Blocking, DCT, to Quantization (DCT&Quant), and the second is RGB-to-YCbCr (RGB2YCbCr). That is, these two hotspots are candidates for acceleration (hereafter referred to as “target hotspots”).

2.2 Motivational Example

In embedded systems, it is essential to design the accelerator(s) of a target application while suppressing an increase in the design time [4]. Conventionally, a practical approach is to design an accelerator for each hotspot and customize a baseline embedded processor with such accelerator(s) to handle each hotspot as a special instruction and execute on its dedicated accelerator. In case of the JPEG compression algorithm, as described in Section 2.1, two accelerators (for DCT&Quant and RGB2YCbCr) are tailored.

For simplicity, we explain an example in Figure 3 to clarify the conventional accelerator designs limitations and our motivation. As illustrated in Figure 3(a), suppose that the computational time of a target application is dominated by two functions (Functions A and B) similarly as the JPEG compression algorithm (i.e., Functions A and B correspond to DCT&Quant and RGB2YCbCr, respectively), area overhead (the left y-axis in Figure 3(b)), speedup (the right y-axis in Figure 3(b)), and energy saving (the left y-axis in Figure 3(c)) are described for three types of conventional accelerator designs targeting both or either of Functions A and B. Accelerators are adopted such that speedup and energy saving can be maximized while satisfying constraints on area overhead (objectives and constraints may change). Since accelerators are likely to become larger for more critical hotspots, trade-offs in area, performance, and energy would be similar to those shown in Figures 3(b) and 3(c). If larger accelerators (i.e., ones for Func.A and Func.A&B) cannot be adopted, then a smaller accelerator (i.e., one for Func.B) will be used.

²Consecutive hotspot functions can be handled as a single hotspot.

The proposed method overcomes these conventional accelerator designs limitations by *introducing an acceptable error* (see the right y-axis of Figure 3(c)). More specifically, our method *reduces computations* by reusing *similar enough* results of previous computations (the details will be provided in the following section), targeting RMS applications that have some error tolerance. Figures 3(b) and 3(c) also describe the trade-offs of the proposed method accelerating *both* Functions A and B³—leveraging our data-reuse mechanism, a single accelerator enables in speeding up *multiple* hotspots only through a small extension in the accelerator’s controller. The major contribution of our method is to *increase the chances of accelerating multiple hotspots*, unlike conventional methods, which require dedicated accelerators for different hotspots.⁴ Our contribution will be significant to overcome the shortcomings of conventional methods when multiple and/or large accelerators cannot be adopted due to stringent design constraints.

Here, we provide some evidential examples to demonstrate the proposed method. In case of the JPEG compression algorithm, a lot of chances of reusing “similar enough results” (hereafter referred to as “approximate data”) exist. For example, RGB2YCbCr and DCT&Quant can reuse previous similar enough results for an average of 48.4% and 29.9%, respectively, even when at most only 32 recent results are preserved for future data reuse. If our accelerator leverages these approximate data in both DCT&Quant and RGB2YCbCr, then an average of $\times 1.22$ and up to $\times 1.45$ speedup can be achieved against the baseline processor with insignificant degradation of the output quality (30.22db in Peak Signal-to-Noise Ratio (PSNR)). However, the conventional design with two accelerators achieves an average of $\times 1.41$ and up to $\times 1.44$ speedup with the PSNR quality of 34.77db⁵. These results are a good evidence of the reasonability and effectiveness of our approach. Our accelerator is expected to bring similar or more benefits to other RMS applications due to chances of data approximation [19].

3 APPROXIMATE DATA REUSE-BASED PROCESSOR

This section presents our proposed method, which reuses approximate data at the architectural level. First, we give an overview of both hardware and software of the entire system, followed by a detailed description of the key modules of our accelerator. Then, we explain how our accelerator is applied to the JPEG compression algorithm.

3.1 Overview

Goal: We propose a novel accelerator with a good trade-off between efficiency and design productivity (or reusability). This is a single accelerator that can be used for speeding up multiple hotspots of a target application or different applications and can achieve sufficient speedup under stringent design constraints. In other words, a single accelerator is shared by multiple hotspots. The key concept is that we introduce an acceptable error in our accelerator design. For RMS applications that have inherent nature of error tolerance, we develop such accelerator by leveraging *data similarity* of inputs fed to target hotspots (functions or loops). In target hotspots, if the current input data are similar enough to previous ones, then their computational results are also expected to be similar enough. This allows the previous results to be reused for the current computations,

³As one may find from Figure 3(c), even if our method can accelerate *both* Functions A and B, the total speedup would be less than the accelerator design of Func. A&B. The reason will be given along with the detailed explanations of the proposed method and evaluations later.

⁴Although some works have implemented a single accelerator targeting multiple applications or functions [3], they only shared some resources (e.g., arithmetic units) between multiple accelerators to force them to implement in one. This can suppress area overhead to some extent, but the extension would not be trivial.

⁵These are a subset of our experiments shown in Section 4, where quantitative results will be fully disclosed.

Fig. 4. The overview of our approximate data reuse-based processor.

leading to skipping computations in the target hotspots. We call such data reuse “*approximate data reuse*” and apply this concept to the accelerator design attached to an existing processor.

Figure 4(a) shows the overview of how our approximate data reuse-based accelerator works. The left and right parts describe the software and hardware flows, respectively. Notations are defined in Figure 4(b). Our accelerator composes of a simple controller and a dedicated memory named a dataset table (DST)⁶ to store limited sets of previous input data and corresponding results of the computations of a target hotspot. In this figure, “*target function or loop*” is a target hotspot (function or loop in most cases) to be accelerated in an application. Each of such hotspots is designated by a set of *check_address* instruction(s) and a *result_address* instruction, starting and ending of the hotspot, respectively. As will be explained in the subsequent section, multiple *check_address* instructions may be available, whereas only one *result_address* instruction can be set. As have been conventionally done [4], we assume that hotspots are determined by profiles and simulation in advance, where the programmers need to annotate a pair of *check_address* instruction(s) and a *result_address* instruction to designate each hotspot.

For simplicity, here we explain the procedure of a single hotspot. Note that multiple hotspots can be handled by adequately setting the DST’s controller and the memory controller, as will be done for the JPEG compression algorithm (the details will be given in the following sections). The entire system works in the four steps as follows:

- (1) The processor performs the application in a normal manner until the program counter (PC) reaches the designated *check_address* instruction(s). Then, the DST’s controller checks if the value(s) of all *check_address* instructions (i.e., *check_value(s)*) is/are registered in the DST. Here, *check_value(s)* is/are a key value to judge the data similarity at the beginning of the target hotspot. In this article, the data similarity is judged by a “threshold” on ignorable LSBs (see Section 3.2).
- (2) If the key value exists in the DST (i.e., “11111” in Figure 4(a)), then the processor is allowed to skip all the instructions of the hotspot by reusing the corresponding resultant data (i.e., *result_value(s)*) whose location in the main memory is specified by *copyfrom*

⁶Hereafter, DST interchangeably represents the entire accelerator and the dedicated memory on the accelerator.

Fig. 5. The entire processor extended with the dataset table (DST) and the memory controller.

(see Section 3.3). Otherwise, the processor is not allowed to skip and hence *dose* process instructions in a normal (precise) manner.

- (3) If instructions are *skipped*, then the result(s) (i.e., `result_value(s)`) is/are reused, if necessary, by being copied to the appropriate location specified by `copyto` (see Section 3.3). Note that, while a data copy is needed by the JPEG compression algorithm, it is not always the case (e.g., applications that only update resultant values repeatedly until convergence). If instructions are *not skipped*, then the current data (i.e., a pair of `check_value(s)` and `copyfrom`) are registered to the DST for future data reuse.
- (4) Finally, the remaining parts of the DST are updated accordingly (see Section 3.2).

In this study, we extend a five pipeline-stage MIPS, which is one of the representative Reduced Instruction-Set Computer (RISC) processors, as depicted in Figure 5. The original MIPS is equipped with only the modules of black arrows and gray boxes, whereas our extensions are depicted as modules of red arrows and yellow boxes. Note that our proposed method can be applied to other types of RISC/CISC processors by adequately setting the DST and the memory controller.

3.2 Dataset Table (DST)

As embedded systems generally have a hardware footprint limitation, the size of DST (i.e., the number of storing datasets) is limited. For efficient data reuse under a limited DST size, we employ the least recently used (LRU) algorithm, which is frequently used for replacing data in caches to efficiently utilize the limited amount of data. As its name represents, the LRU algorithm replaces the least recently used data by the new data. We follow this approach to select the data to be replaced for the DST update.

3.2.1 Structure. Figure 6 describes an example of the structure of our DST. The notations used in Figure 6 are defined in Table 1. This DST can store up to four datasets, each of which is composed of a tag flag, a valid flag, `check_value(s)`, and a `copyfrom_top` address. Here, we explain the major components of the DST (in the lower part of Figure 6):

- **tag:** The tag flags memorize the order of the latest updates of the rows. The smaller the tag value, the more recent it has been accessed (used or overwritten).
- **valid:** All valid flags are initialized with 0 until their corresponding row is filled with a set of data for reuse (the valid flag is updated to 1). This means that data replacement happens only after all tag flags are set to 1. All valid flags are reset to 0 every time another hotspot is targeted.
- **check_value(s):** This is the input of the target hotspot to construct a key value of the DST. This example shows the case of assigning four check_values for each data reuse. The number of check_values varies by applications.

Fig. 6. DST structure.

Table 1. Notations Introduced in Figure 6 and Their Description (Notations Already Defined in Figure 4(b) Are Omitted)

Notation	Description
PC	Program counter
WBvalue	Result of an arithmetic operation or load operation. This is extracted from the write-back (WB) stage of the processor and used for check_value(s).
memaddr	Memory address to/from which the data is copied (i.e., copyto_top /copyfrom_top). This is designated by a memory access instruction and extracted from the decode (ID) stage of the processor.
dnum	The amount of result_value(s).
dintr	Interval between result_value(s) stored in the main memory.
dwidth	Data width of result_value(s) (byte, half word, or word).
skip	Flag given to the CPU. Set to 1 if instructions are skipped, otherwise 0.
reuse	Flag given to the memory controller. Set to 1 if result_value(s) are reused, otherwise 0.
state	State register to handle the data reuse/replacement and set adequate control signals.
address settings	Module to set the adequate address for data reuse and DST update, depending on the target hotspot being processed.
control signals	Module to generate the control signals inside of the DST.
data copy settings	Module to set adequate control signals to the memory, depending on the hotspot being processed.
check_values	Temporal register to store all check_value(s) until they are all compared with the data in the DST (for data reuse) or registered (for data replacement).

Inputs, outputs, and modules are summarized at the top, middle, and bottom chunks, respectively.

Fig. 7. DST update.

Fig. 8. Timing diagrams of the DST.

- **copyfrom_top**: This indicates the top address of the resultant (reused) data in the main memory. In the JPEG compression, it is set to the first data location of RGB pixel data and that of an 8×8 pixel block for RGB2YCbCr and DCT&Quant, respectively (see Section 3.4). Contrary to check_value(s), which may be single or multiple, a single copyfrom_top address (i.e., only the top address) should be registered. Data to be copied and the destination (the top address) are designated by result_address, copyto_top, dnum, dintr, and dwidth.

3.2.2 Behavior. The controller (in the upper part of Figure 6) is clock-gated to work only when the PC reaches check_address and copyfrom_top for beginning and completing instruction skipping, respectively. Here, we explain the behavior of the DST in Figure 7. Also, the timing diagrams are described in Figure 8, where the compare, update, and update_tag represent the comparison of check_value(s) in the temporal register and the memory in the DST, a flag for data replacement, and a flag for the tag update, respectively (these signals are omitted from Figure 7).

- **In case of data reuse** (the bottom left of Figure 7): Let us assume that the data in the second row are reused. Since the second row is now the most recently accessed and used, the tag flag is updated to 0, indicating the most fresh data. All rows with the smaller tag

value are also updated while keeping the same order. Note that the contents of the second row (i.e., `check_values` and `copyfrom_top`) are not updated. That is, when the data are reused, only the tag flags are updated. In our implementation, the current `check_value(s)` in the temporal register is/are compared with all the `check_values` in the DST in parallel (comparators are omitted from the figure). Then, the skip flag and the reuse flag are set to 1 and fed to the memory controller (explained in Section 3.3) with the other output signals. The timing diagram for this procedure is displayed in Figure 8(a). The skip flag is set as soon as reusable data is found to exist. In the next cycle, the reuse flag is set to let the memory controller reuse the data, and tag values are all updated.

- **In case of no data reuse** (the bottom right of Figure 7): If no data can be reused after the processor performs the precise calculation by computing all of the instructions in the target hotspot, then a set of `check_value(s)` and the corresponding `copyfrom_top` is registered. Since the DST is already full, the LRU set of data (i.e., one with the largest tag value) is replaced by the new set of data as revealed in the first row. After the update, the tag value of the first row is set to 0, meaning that it is now the most fresh. The other tag values are incremented by 1. The timing diagram for this case is displayed in Figure 8(b)—the difference from Figure 8(a) is that the update flag is set when the computations in the hotspot are all finished.

Judgment of “approximate data reuse” is defined by the matching degree between the current `check_value(s)` and `check_value(s)` in the DST. We regard this as “matched” if the current `check_value(s)` *almost* match(es) with a set of `check_value(s)` of the same row in the DST’s memory—we introduce a “threshold” as how many LSBs can be ignored to judge the *almost* match. For example, if `check_value(s)` is/are 8 bits and the threshold is set to 2, only `check_value(s)[7:2]` (6 bits) are compared. Then, if four `check_values` are used, then each row “`check_values`” is composed of 24 bits. As shown in the experimental results, the DST footprint can vary by the DST size (i.e., the number of reusable data) and the threshold.

As aforementioned, multiple hotspots can be handled by a single DST through setting the DST controller adequately. Because each hotspot is designated by a pair of `check_address` and `result_address`, the hotspot currently handled can be identified with the PC. Then, all data in the DST are initialized and updated every time the hotspot switches are processed to make each hotspot utilize its *own* previous results. In other words, multiple hotspots share the DST’s memory. This approach can effectively mitigate the circuit overhead, but it overwrites the data with the results of another hotspot and may sacrifice some chances of reusing data. For example, in the JPEG compression, since two hotspots, RGB2YCbCr and DCT&Quant, are performed at every 16 lines, the results of the previous 16 lines are no longer available when the next 16 lines are processed. However, this effect is trivial considering the practical size of the DST for embedded systems (i.e., realistically, the DST is not large enough to keep the results of the previous 16 lines).

3.3 Memory Controller

If instructions in a target hotspot are skipped only, when successive operations access the results of the target hotspot, then their memory locations would contain no valid data. Thus, the corresponding result(s) (i.e., `result_value(s)`) need(s) to be compensated appropriately. If the amount of `result_values` is small, then the data may be directly registered in the DST. Otherwise, two possible approaches can be considered: data reference and data copy. For each of invalid data, the first approach (the data reference) memorizes the memory address of the reused approximate data by using an additional circuit and on-chip memories such as content addressable memory. Because the frequency of data reuse is unexpected at design time and is totally data-dependent,

Fig. 9. Memory controller.

this extension has to be applied to the *whole* memory, resulting in a large circuit overhead. Also, using on-chip memories would be costly if the reused data size is large. Consequently, we adopt the latter approach, *data copy*, which requires memorization of only the top memory address (i.e., `copyfrom_top`) in the DST as seen in Figure 6. This approach is scalable as it can be applied irrespective of the number of `result_values`. As explained below, the data copy is handled by the extended memory controller in four steps.

Like most processors, MIPS (TigerMIPS [24]) uses a two-port memory and takes one cycle for read and write. Hence, data copy from one location to another in the memory takes two cycles in total. The overall structure of the memory controller is depicted in Figure 9, using notations that are described in Table 2. In our work, the memory controller leverages the time that the memory is not accessed by the CPU so that the performance degradation can be minimized by hiding the data copy latency. In other words, the computation in the CPU and data copy can be performed in parallel. By this parallelism, as will be discussed later, we observe that the latency of data copy can be totally hidden in our case study. Similarly, latency hiding can be also expected for other realistic applications.

- (1) **Initialization:** Only when the instructions skipping is performed and the PC reaches the `result_address` (i.e., `reuse = 1` and `PC = result_address`), the `act` flag is set to 1, which is kept until the copy of `result_value(s)` is done. Also, `counter`, `copyfrom_top`, and `copyto_top` are initialized as described in Table 2.
- (2) **Data load:** The `result_value(s)` is/are loaded from and stored in the memory alternately for data copy. First, the `result_value` is loaded from the memory if `act = 1`, `we = 0`, and `mem_access = 2'b00`. The loaded data is temporarily stored in `tempdata`. Then, the signal (`we`) and the register (`copyfrom`) are updated as `we = 1` and `copyfrom += dintr`, respectively.
- (3) **Data store:** The data in `tempdata` is copied back to the appropriate location indicated by `copyto` if `act = 1`, `we = 1`, and `mem_access = 2'b00`. Then, similarly to the step (2), the signal (`we`) and the register (`copyto`) are updated as `we = 0` and `copyto += dintr`, respectively.
- (4) **Counter update:** Every time one data is copied, the counter is incremented by 1. Then, when the counter reaches the maximum bound (i.e., `dnum - 1`), meaning that a set of

Table 2. Notations Used in Figure 9 and Their Description (Notations Previously Defined in Figure 4(b) and Table 1 Are Omitted)

Notation	Description
mem_access	Flag to identify if the CPU accesses the memory: 2'b00 for no read or write, 2'b01 for write, and 2'b10 for read (2'b11 is not used).
width_in	Data width (byte, half word, or word).
write_in	Flag to enable the memory write.
addr_in	Memory address to write or read.
data_in	Data to be written to the memory. This indicates result_value during the data copy.
readdata	Data to be read from the memory. This indicates result_value during the data copy.
width_out	Data width. It is set to half word if data copy is done, otherwise width_in.
write_out	Flag, which is set to we if data copy is done (i.e., load result_value from the memory if write_out = 0, and store result_value to the memory if write_out = 1), otherwise write_in.
addr_out	Memory address to write or read. This is set to the next copyfrom to load result_value or the next copyto to store result_value if data copy is done, otherwise addr_in.
data_out	Data to write to or read from the memory.
act_gen	Module to generate the act flag to activate data copy. The act flag is set to 1 if reuse = 1 and PC = result_address, otherwise 0.
we	Flag, which is set to 0 to load data, otherwise 1 to store data.
copyfrom	Register to have the memory address from which result_value is currently loaded. This is initialized by copyfrom_top when a data copy starts, and incremented by dintr to point to the address from which the next result_value is loaded.
copyto	Register to have the memory address in which data is currently stored. This is initialized by copyto_top when data copy starts, and incremented by dintr to point to the next address in which the next result_value is stored.
counter	Module to count the amount of copied data up to the maximum bound (2 for RGB2YCbCr and 63 for DCT&Quant in this article). This is initialized to 0, and incremented by 1 every time one result_value is stored.
tempdata	Temporal register to store result_value loaded from the memory.

Inputs, outputs, and modules are categorized at the top, middle, and bottom chunks, respectively.

reused data is all copied, it is initialized as 0. Additionally, the act flag is set to 0 to let the memory controller understand that the data copy is done.

As aforementioned, the data copy is not always done ceaselessly, because its procedure may be *interrupted* if the CPU requires a memory access. In other words, we prioritize the memory access by the CPU over the data copy. The memory access by the CPU can be identified by mem_access (2'b01 or 2'b10). If the CPU accesses the memory while the data copy is being done, then the data copy is immediately interrupted and restarts from the next procedure as soon as the CPU completes the memory access. The memory access signals from the CPU are only propagated to the memory through the memory controller (i.e., width_out = width_in, write_out = write_in, addr_out = addr_in, and data_out = data_in). In this way, the data copy can be done in parallel with the CPU computations, and its latency can be totally hidden in our case study. Such latency hiding can be also expected in other applications.

For simplicity, the above explanations assumed the case that the baseline processor has no memory hierarchy. If L1 caches for instructions and data (I-cache and D-cache) are used, similarly to some processor-accelerator systems such as Reference [2], while the I-cache will fit within the

Fig. 10. DST settings for RGB2YCbCr and DCT&Quant.

processor, because it will be accessed by the processor only, then the D-cache will be shared between the processor and the accelerator to speedup data accesses from both of them. Recall that unlike conventional precise computing-based accelerators, our DST does not perform computations of the target hotspots and only decides if similar enough previous data would be reused. Therefore, when reusing a previous data, our DST will directly activate the memory controller by the reuse signal to conduct the data copy as explained above. Although this may affect the cache hit rate, the speedup effect of skipping instructions is expected to be larger than the cache miss penalty, particularly for large hotspots.

3.4 Employment to JPEG Compression

In this article, we aim to accelerate two hotspots (i.e., RGB2YCbCr and DCT&Quant) using a single DST for the JPEG compression algorithm. For each hotspot, we first find check_addresses and result_address through instruction-set simulation (see our experimental setup in detail in the next section). As described in Section 3.3 and Figure 6, the pixel positions of check_values are obtained from the WBvalues of the instructions designated by check_addresses (the details of check_addresses setting are provided below). We let the DST have four check_values to construct a key value, and set check_values and copyfrom_top as described in Figure 10:

- **RGB2YCbCr:** The addresses of the three load instructions to get an RGB value were used for check_addresses, and hence their values were check_values. Because we used only three check_values for RGB2YCbCr, we set a dummy value for the fourth check_value in each row. Three converted YCbCr values were used for result_values, and the first address of those three store instructions was used for copyfrom_top.
- **DCT&Quant:** Although an 8×8 block had 64 pixels, we used the addresses of four load instructions to get four corner pixels of the block for check_addresses (highlighted as red pixels in Figure 10). The DST can skip instructions only after checking the last check_value. Therefore, during the software compilation, we changed the instructions order by loading four check_values first to maximize the speedup effect when reusing data (i.e., as many instructions as possible can be skipped). Although this is against the spatial data locality and may thus affect the efficiency of memory access for a general cache memory, this DST-oriented optimization enhances the benefit of instructions skipping by the DST more than the cache miss penalty. An quantized 8×8 block was used for result_values, and the address of a store instruction for the top left pixel of the quantized block was used for copyfrom_top.

Although the number of check_values in the DST is determined by the maximum number of check_values among the target hotspots, thresholds and the numbers of check_values to be used can be set differently for different hotspots. Such different settings on the threshold and

Fig. 11. Different numbers of pixels (i.e., check_values) to construct a key value.

Fig. 12. Trade-offs for different check_values.

Fig. 13. Visual comparisons of output images “kodim23” by different number of check_values and the two LSB thresholds.

check_values can flexibly let a single DST accelerate multiple hotspots that process different data (e.g., an RGB value in RGB2YCbCr and an 8×8 block in DCT&Quant) and have different error propagation/masking effects (e.g., while DCT&Quant naturally produces some errors through the image compression, RGB2YCbCr does not produce errors). Hence, this achieves a good trade-off between the accuracy and speedup.

As we use only four pixels values to construct a key value for DCT&Quant, even if the threshold of ignorable LSBs is set to 0, approximate data reuse is performed. Then, one may wonder if using four corner pixels would be aggressive or conservative. As illustrated in Figure 11, we have quantitatively evaluated different numbers of pixels to construct a key value (three, four, six, and eight) to demonstrate the reasonability of selecting these four pixels and the extensibility of the DST. As revealed in Figure 11, we selected pixels far away from each other, since neighboring pixels tend to have similar values. Figure 12(a) describes the trade-offs between the accuracy (in terms of PSNR; x-axis) and speedup against the baseline processor (y-axis) varying the number of check_values and the two LSB thresholds. Also, Figure 13 describes the visual comparison of output images for different number of check_values (the DST size was 32, and the LSB thresholds on RGB2YCbCr (T_{color}) and DCT&Quant (T_{block}) were set to 4 and 2,

Table 3. Experimental Environment and Parameter Settings

High-level synthesis tool	eXCite 5.2n
Logic synthesis tool	Design Compiler F-2011.09-SP2
Software compiler	mipsel-linux-elf-gcc-4.6.4
Clock frequency	500MHz
Memory model	CACTI 5.3 [12]
ASIC process technology	Nangate 45nm
DST size (#DST)	1, 2, 4, 8, 16, 32, 64
LSB threshold of RGB2YCbCr (T_{color})	0, 1, 2, 3, 4
LSB threshold of DCT&Quant (T_{block})	0, 1, 2, 3, 4

respectively). Clearly, fewer check_values lead to poorer accuracy by more aggressive data reuse, whereas more check_values lead to better accuracy by more conservative data reuse. As PSNR is expected to be in the range of 30–50dB for images [10], using only three check_values may not be satisfiable in most cases. Contrarily, using eight check_values would sacrifice the speedup and the area overhead (Figure 12(b) shows the area overhead for different check_values varying the two thresholds ($T_{color}=T_{block}$); the overhead is proportional to the number of check_values). Consequently, we have concluded that using four check_values is in the best trade-off between speedup, circuit area, and accuracy. Through these evaluations, we have confirmed that the DST is tunable and applicable to different applications and/or design constraints.

Although the speedup effects by different number of check_values and software refinement (e.g., instructions order) would depend on the application, the attempts of reducing the number of check_values and enlarging the distance from check_address to result_address are both essential and commonly beneficial to other applications. These should be done by the help of profiles and statistical analysis [4].

4 EVALUATION

This section shows the effectiveness of our proposed method over conventional ones through a case study using the JPEG compression algorithm.

4.1 Experimental Setup

Our case study was conducted on the JPEG compression algorithm to demonstrate the effectiveness of our work. TigerMIPS [24],⁷ composed of five pipeline stages, was used as the baseline processor. We assume that the baseline MIPS uses an ALU, a multiplier, and a divider as the basic computational resources in the execution (EX) stage. For simplicity, no memory hierarchy is assumed (i.e., no cache).⁸ The environment used (i.e., tools, library, and constraints) are summarized in the upper part of Table 3.

In our evaluation, we compared the following methods:

- **TigerMIPS:** The baseline MIPS, denoted as *TigerMIPS* hereafter.
- **Conventional accelerators:** Conventional, precise accelerators targeting hotspots. Here, we realized three types of accelerators, *RGB2YCbCr_c*, *Quant&DCT_c*, and *All_c*, which

⁷Because the original TigerMIPS written in Verilog HDL was designed for FPGAs and hence utilizes FPGA built-in resources, we modified the description by removing the use of such resources to make it synthesizable for ASIC.

⁸It should be noted that the speedup effect of our work is expected to increase by having the I-cache and D-cache, since those caches will reduce the total cycle counts on both of the baseline processor and our work.

Fig. 14. Circuit area.

accelerate RGB2YCbCr, DCT&Quant (from 8×8 Blocking, Quantization to DCT), and both, respectively—recall that, as shown in Figure 2, DCT&Quant and RGB2YCbCr are the first and second hotspots. The accelerators were implemented by high-level synthesis.

- **Ours:** Our proposed method leveraging the DST-based approximate data reuse. The extensions explained in Section 3 were all done at the register transfer level using Verilog HDL. The target hotspots are both RGB2YCbCr and DCT&Quant. As our method can vary with the DST size, hereafter our method is denoted as $\#DST$, where $\#$ represents the DST size. Similarly, the LSB thresholds (T_{color} and T_{block}) can be individually varied. We set seven and five different parameters for the DST size and the LSB thresholds, respectively, as tabulated in the lower part of Table 3. We exhaustively evaluated 35 implementations to quantitatively discuss good/bad combinations of parameter settings.

We evaluated all of the above methods in terms of circuit area, performance (cycle counts; as all implementations work under 500MHz), and energy consumption. Also, for a comprehensive evaluation, we introduced the product of circuit area, performance, and energy (larger scores represent the better results or better “efficiency”). Additionally, only for our method ($\#DST$), the accuracy was evaluated in terms of PSNR against the precise results obtained by *TigerMIPS* and three conventional accelerators. We used all the 24 images from Kodak Lossless True Color Image [11].

4.2 Experimental Results

The results of all methods in terms of circuit area, cycle counts, energy saving, and comprehensive evaluation depend on the input images, and thus are described as bar graphs in Figures 14, 15, 16, and 17, respectively. All these metrics are taken in the y-axis and represent better results in smaller values. The results in Figures 16 and 17 are normalized by those of *TigerMIPS*.

With the reference to the results of circuit area in Figure 14, conventional methods considerably increase the circuit area for more dominant hotspots as pointed out in Section 2.2. In some cases, due to design constraints, it would be difficult to adopt *DCT&Quant_c* and *All_c* whose area is approximately $\times 1.59$ and $\times 1.63$ of *TigerMIPS*, respectively. Contrarily, our method can successfully mitigate the circuit area overhead in most parameter settings. Our method increases the circuit overhead by increasing $\#DST$ and decreasing the thresholds, since they determine the size of the memory and comparators in the DST. Compared with conventional methods, for all DST sizes other than 64, the area is less than *DCT&Quant_c* and *All_c*. From these results, we can confirm that our method leads to small extensions to the baseline processor and would be applicable even to stringently constrained embedded systems by tuning the parameters.

Cycle counts in Figure 15 for the *entire* application execution are depicted in the form of rectangles (to clarify the minimum and maximum counts), because for all methods the cycle counts

Fig. 15. Total cycle counts for the entire application execution.

Fig. 16. Energy saving.

Fig. 17. Comprehensive evaluation (efficiency).

depend on the image type and size. Our method is further affected by the frequency of data reuse, which is totally data-dependent, even for the same combination of parameter settings. Conventional methods achieve good cycles reduction, which becomes larger by targeting more hotspots. As compared with these methods, our method achieves moderate cycles reduction for smaller thresholds and smaller #DST but almost comparable or even more cycles reduction for larger thresholds and larger #DST. This is because while conventional methods *always* accelerate the target hotspots, our method can do only when the conditions of data reuse are satisfied.

As the energy consumption is affected by both the circuit area and cycle counts, as expected, the results of energy consumption in Figure 16 have similar tendencies like those of cycle counts in Figure 15. For the results of our method, the energy efficiency in Figure 16 is improved more than cycle counts reduction in Figure 15 because of the area suppression under most combinations of parameter settings (recall Figure 14). More specifically, while our method achieves moderate energy saving in most combinations of parameter settings, our method still hits the lowest energy results under larger thresholds, indicating that, for some images, it enhances more energy saving than *All_c*.

To summarize the above results, we comprehensively compare all methods as shown in Figure 17. In spite of large cycles reduction, conventional methods achieve modest scores because of a large area overhead. The efficiency of our method differs in #DST and thresholds—it gradually improves by increasing #DST until #DST = 16 and degrades when #DST > 16. When #DST = 64, the efficiency is rather lower than *TigerMIPS* in most cases because of a large circuit overhead (recall Figure 14). Overall, irrespective of the setting of T_color , when #DST = 8 or 16 and $T_block = 3$ or 4, our method achieves *equivalent or better efficiency* than *All_c*.

Figure 18 describes the effect of approximate data reuse in our method. The y-axis shows the PSNR of the final output images against the reference (the precise results by *TigerMIPS* and the conventional methods). The results are shown by bar graphs to reveal the maximum and minimum PSNR values. Also, for visually comparing the results, some selected output images, whose PSNR is almost around the average among 24 images, are shown in Figure 19. As expected, while the results with conservative data reuse (i.e., smaller #DST and/or smaller thresholds) show less degradation, those with aggressive data reuse (i.e., larger #DST and/or larger thresholds) have larger degradation. Recall that PSNR for images is expected to be in the range of 30–50dB [10], overall (from further comprehensive comparisons through Figures 17 and 18), we conclude that #DST = 8 or 16, $T_color \leq 3$, and $T_block = 3$ are the well-balanced parameter settings for image compression.

Discussions on parameter settings: Here, we reveal findings on the parameter settings through exhaustive evaluations.

- (1) As aforementioned, in our case study, #DST = 8 or 16, $T_color \leq 3$, and $T_block = 3$ are the well-balanced parameters, and with these parameters, our method achieves the better efficiency than conventional accelerator designs.
- (2) For #DST ≥ 32 , our method degraded the efficiency, since cycles reduction and energy saving were canceled out by the area overhead. From another parameter's perspective, thresholds also affect all metrics but less significantly than #DST. Although the effects of specific parameter settings should differ from different applications and data to be processed, it can be commonly said that #DST should be determined more carefully than thresholds.
- (3) #DST and thresholds should be determined to encourage each other and avoid the contradiction on the efforts of data reuse. As shown in Figure 17, although relaxing both of them is likely to raise the peak (or maximum) efficiency, excessive #DST rather degrades

Fig. 18. PSNR ranges of output images.

Fig. 19. Visual comparisons of output images.

Fig. 20. Comparison on data reuse rates of RGB2YCbCr and DCT&Quant.

the efficiency due to a large area overhead. Similarly, strict thresholds like $T_block = 0$ also disturb the data reuse and cannot perform instructions skipping.

- (4) Thresholds of different hotspots have different impacts on the frequency of data reuse. In the JPEG compression, the data reuse frequency largely differs between RGB2YCbCr and DCT&Quant even with the same LSB thresholds setting (i.e., $T_block = T_color$), as shown in Figure 20. This is affected by how check_values are referred and selected in each hotspot. Recall the algorithmic flow and the DST settings described in Figures 1 and 10, RGB2YCbCr is performed consecutively for neighboring 16 lines, and thus is likely to refer to previous results in the DST and reuse them. On the other hand, DCT&Quant is performed for three channels individually almost in a round-robin manner, and thus can refer to previous results less frequently. Furthermore, DCT&Quant selects check_values at a distance, resulting in lower frequency of referring to previous results due to more diverse key values. To increase the chances of data reuse, DCT&Quant needs more

aggressive thresholds than RGB2YCbCr. To sum up, the algorithmic flow of the target application (e.g., how hotspots are executed) largely affects check_values reference and the significance of thresholds, and thus needs to be taken into account.

For simplicity and comprehensibility, we targeted the JPEG compression algorithm in this article and revealed that introducing the concept of “approximate computing (AC)” in the accelerator design can break through conventional designs and widen the design space exploration worthfully. Considering that RMS applications, which are intensively studied these years, have more error tolerance, more aggressive approximate data reuse is expected to be acceptable, leading to further enhance the efficiency of our proposed method. For example, face detection by machine learning may perform well enough for images with even less than 30dB. Widening the scope of applications and revealing the merits/limitations of our DST-based accelerators is the subject of our future work.

5 CONCLUSIONS

This article proposed a novel accelerator design method that achieves sufficient speedup and energy saving under stringent area constraints. Our work employs “approximate data reuse,” one of AC techniques, focusing on the fact that in many RMS applications, some computational results can be reused if input data that are similar enough to the current input data have been used for recent computations. By leveraging this feature, our accelerator, composed of a set of memories and a simple controller, can accelerate multiple parts (i.e., functions or loops) of an application, unlike conventional designs that require separate accelerators. Our proposed method is realized by both hardware (architecture) and software (compilation) techniques and helps to effectively reduce computations by skipping corresponding instructions. Through intensive evaluations in the case study using a realistic application (image compression), we demonstrated the effectiveness of our work over conventional accelerator designs in terms of not only some design metrics (area, speedup, and energy) but also extensibility to other applications and/or design constraints. Also, we revealed important findings that will lead to efficient exploration on combinations of key parameter settings for other applications.

REFERENCES

- [1] S. T. Chakradhar and A. Raghunathan. 2010. Best-effort computing: Re-thinking parallel software and hardware. In *Proceedings of the Design Automation Conference*. 865–870.
- [2] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. 2012. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*. 17–24.
- [3] N. Engelhardt. 2010. Application of multimode HLS techniques to ASIP synthesis. In *INRIA-IRISA Rennes Bretagne Atlantique, Équipe Cairn*.
- [4] C. Galuzzi and K. Bertels. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfig. Technol. Syst.* 4, 2, Article 18, 18:1–18:28.
- [5] A. Ghofrani, A. Rahimi, M. A. Lastras-Montano, L. Benini, R. K. Gupta, and K. Cheng. 2016. Associative memristive memory for approximate computing in GPUs. *IEEE J. Emerg. Select. Top. Circ. Syst.* 6, 2 (June 2016), 222–234.
- [6] X. He, W. Lu, G. Yan, Y. Han, and X. Li. 2017. Exploiting the potential of computation reuse through approximate computing. *IEEE Trans. Multi-Scale Comput. Syst.* 3, 3 (2017), 152–165.
- [7] N. M. Ho, E. Manogaran, W. F. Wong, and A. Anoosheh. 2017. Efficient floating point precision tuning for approximate computing. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 63–68.
- [8] M. Imani, D. Peroni, and T. Rosing. 2017. CFPU: Configurable floating point multiplier for energy-efficient computing. In *Proceedings of the Design Automation Conference*. 76:1–76:6.
- [9] M. K. Jain, M. Balakrishnan, and A. Kumar. 2001. ASIP design methodologies: Survey and issues. In *Proceedings of the International Conference on VLSI Design*. 76–81.
- [10] J. Kaur and B. Singh. 2014. Comparison of LSB and predictive coding using PSNR and MSE. *Int. J. Comput. Appl.* 98, 7 (2014), 35–39.
- [11] Kodak. 2013. Kodak lossless true color image. Retrieved from <http://r0k.us/graphics/kodak/>.

- [12] HP Labs. 2010. CACTI 5.3. Retrieved from <http://quid.hpl.hp.com:9081/cacti/>.
- [13] S. Lee, L. K. John, and A. Gerstlauer. 2017. High-level synthesis of approximate hardware under joint precision and voltage scaling. In *Proceedings of the Design, Automation Test in Europe*. 187–192.
- [14] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. 2015. Doppelgänger: A cache for approximate computing. In *Proceedings of the International Symposium on Microarchitecture*. 50–61.
- [15] J. S. Miguel, M. Badr, and N. E. Jerger. 2014. Load value approximation. In *Proceedings of the International Symposium on Microarchitecture*. 127–139.
- [16] H. Osawa and Y. Hara-Azumi. 2017. Approximate data reuse-based processor: A case study on image compression. In *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia*. 32–40.
- [17] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. 2012. Programming with relaxed synchronization. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*. 41–50.
- [18] M. C. Rinard. 2007. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. 369–386.
- [19] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. 2013. Approximate storage in solid-state memories. In *Proceedings of the International Symposium on Microarchitecture*. 25–36.
- [20] D. Sengupta, F. S. Snigdha, J. Hu, and S. S. Sapatnekar. 2017. SABER: Selection of approximate bits for the design of error tolerant circuits. In *Proceedings of the Design Automation Conference*. 72:1–72:6.
- [21] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel. 2015. A low latency generic accuracy configurable adder. In *Proceedings of the Design Automation Conference*. 86:1–86:6.
- [22] Y. Tang, Y. Wang, H. Li, and X. Li. 2017. ApproxPIM: Exploiting realistic 3D-stacked DRAM for energy-efficient processing in-memory. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 396–401.
- [23] Y. Tian, T. Wang, Q. Zhang, and Q. Xu. 2017. ApproxLUT: A novel approximate lookup table-based accelerator. In *Proceedings of the International Conference on Computer-Aided Design*. 438–443.
- [24] University of Cambridge. 2010. The tiger “MIPS” processor. Retrieved from <http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html>.
- [25] Y. Wang, H. Li, and X. Li. 2017. Real-time meets approximate computing: An elastic CNN inference accelerator with adaptive trade-off between QoS and QoR. In *Proceedings of the Design Automation Conference*. 33:1–33:6.
- [26] Q. Xu, T. Mytkowicz, and N. S. Kim. 2016. Approximate computing: A survey. *IEEE Design Test* 33, 1 (2016), 8–22.
- [27] T. Yamamoto, I. Taniguchi, S. Yamashita, H. Tomiyama, and Y. Hara-Azumi. 2017. A systematic methodology for design and worst-case error analysis of approximate array multipliers. *IEICE Trans. Fundam. Electron.* E100-A, 7 (2017), 1496–1499.
- [28] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi. 2013. Approximate XOR/XNOR-based adders for inexact computing. In *Proceedings of the International Conference on Nanotechnology*. 690–693.
- [29] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. 2014. ApproxIt: An approximate computing framework for iterative methods. In *Proceedings of the Design Automation Conference*. 1–6.

Received January 2019; revised May 2019; accepted June 2019