Evaluating the Potential Applications of Quaternary Logic for Approximate Computing

CHRISTOS SAKALIS, ALEXANDRA JIMBOREAN, and STEFANOS KAXIRAS, Uppsala University, Sweden
MAGNUS SJÄLANDER, Uppsala University, Norwegian University of Science and Technology, Norway

There exist extensive ongoing research efforts on emerging atomic-scale technologies that have the potential to become an alternative to today's complementary metal–oxide–semiconductor (CMOS) technologies. A common feature among the investigated technologies is that of multi-level devices, in particular, the possibility of implementing quaternary logic gates and memory cells. However, for such multi-level devices to be used reliably, an increase in energy dissipation and operation time is required. Building on the principle of approximate computing, we present a set of combinational logic circuits and memory based on multi-level logic gates where we can trade reliability against energy efficiency. Keeping the energy and timing constraints constant, important data are encoded in a more robust binary format while error-tolerant data are encoded in a quaternary format. We analyze the behavior of the logic circuits when exposed to transient errors caused as a side-effect of this encoding. We also evaluate the potential benefit of the logic circuits and memory by embedding them in a conventional computer system on which we execute jpeg, sobel, and blackscholes approximately. We demonstrate that blackscholes is not suitable for such a system and explain why. However, we also achieve dynamic energy reductions of 10% and 13% for jpeg and sobel, respectively, and improve execution time by 38% for sobel, while maintaining an adequate output quality.

CCS Concepts: • **Hardware → Analysis and design of emerging devices and systems**;

Additional Key Words and Phrases: quaternary computing, approximate computing, circuits, simulation

ACM Reference Format:
Christos Sakalis, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Evaluating the Potential Applications of Quaternary Logic for Approximate Computing. *ACM J. Emerg. Technol. Comput. Syst.* 0, 0, Article 0 (2019), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In approximate computing, power, performance, and reliability are traded one for another by relaxing the precision constraints of computing, storing, and communicating data [33]. A wealth of applications have, by their nature, approximate inputs or outputs, or include the notion of approximation in their core algorithms. This limits the need for exact computation, storage, and communication, insofar as a system's architecture can exploit such knowledge. Such applications can be commonly found in big data analytics, image processing, sensor applications, and even scientific workloads such as simulations.

This work was funded by Vetenskapsrådet project 2015-05159. The computations were performed on resources provided by SNIC though UPPMAX and by UNINETT Sigma2.
Authors' addresses: Christos Sakalis; Alexandra Jimborean; Stefanos Kaxiras, Uppsala University, Box 337, Uppsala, 75105, Sweden, [firstname.lastname]@it.uu.se; Magnus Själander, Uppsala University, Norwegian University of Science and Technology, Sem Sælandsvei 9, 7491, Trondheim, Norway, magnus.sjalander@ntnu.no.

1550-4832/2019/0-ART0 $15.00
https://doi.org/0000001.0000001

In addition to working with approximate inputs and algorithms, several proposed approaches rely on automatic software transformations that trade accuracy against computational effort, e.g., by skipping tasks, loop perforation, sampling reduction of inputs, or even selecting a different algorithm [3, 52]. Similarly, application specific integrated circuits (ASICs) are often designed considering the required precision for acceptable quality of service (QoS). In signal processing, for example, the bit-width of various stages (ADC, filters, etc.) are adjusted to minimize the computational effort while meeting signal to noise ratio requirements. General-purpose processors, on the other hand, are commonly designed with absolute accuracy for the maximum precision of the system, e.g., 32 or 64 bits for all integer operations. This makes it impossible to take advantage of the relaxed precision requirements in the applications in order to improve energy efficiency and performance. Several works have proposed circuits that trade performance and energy against accuracy, for example, by designing circuits that are simplified and, thus, generate the wrong result for specific inputs [1] or through automatic pruning or substitution of the circuit [26, 49]. Such solutions create circuits that are purely approximate and require additional precise circuits in the system. For example, address calculations cannot be performed on an approximate adder, so an additional precise adder is required.

Based on our earlier work on multi-level logic circuits [44] and caches [45], we propose a set of logic circuits and memory that can modulate their precision while keeping the area overhead low. The multi-level behavior shown in Fig. 1 naturally appears in emerging devices such as single electron (SET) and single atom transistors (SAT). The inherent multi-level characteristic of these devices enables the design of complex multi-level operations such as a half adder consisting of only two devices [35]. Our proposed logic circuits and memory are based on such emerging multi-level technologies, specifically quaternary (four-level) devices. For this work, we assume the existens of basic quaternary logic gates (e.g., And, Xor, Fa, etc.) and memory cells based on such quaternary devices. We observe that the computational accuracy can be traded against power and performance simply by encoding data either in a binary or quaternary format (Sec. 3). Treating the data as binary requires more gates, wires, and memory cells, as each digit contains less information, but the error resilience of the devices is increased because storing a single bit results in more distinct and noise tolerant signal levels. On the other hand, treating the data as quaternary reduces the required number of gates, wires, and memory cells, but also reduces the error resilience. We use quaternary instead of any other multi-level encoding because it offers a balance between error resiliency, circuit complexity, and energy efficiency. Using an encoding that is not a power of two would complicate parts of the combinational logic circuits, and the next larger power of two (eight) is impractically large. In this work:

(1) We propose proof-of-concept quaternary designs for the most common combinational logic circuits, namely a multiplier (Sec. 4.1.2), and a shifter (Sec. 4.1.3). These are supplemented by a parallel adder (Sec. 4.1.1) that is part of our previous work.
(2) We evaluate the potential output errors (Sec. 5.1, 5.3) and energy gains (Sec. 5.4) of these designs, while trying to be as technology agnostic as possible.
(3) We evaluate the error tolerance (Sec. 6.2) and potential energy usage and performance (Sec. 6.3) of three benchmark applications, `jpeg`, `sobel`, and `blackscholes`.

Our goal for the evaluation is to be as technology agnostic as possible. Multi-level transistor technologies are still in their infancy and it is impossible to predict which, if any, the industry will adopt. To this extent, we do not use specific size, energy, and delay characteristics in our evaluation but instead focus on the higher level comparison between binary and quaternary designs. To the best of our knowledge, this is the first work that not only describes the logic-level designs but also evaluates, in detail, their potential adoption on conventional computer systems.

2 RELATED WORK

There is a significant amount of work on approximate circuit design using CMOS technologies. Voltage overscaling can be used to reduce energy usage when absolute precision is not required [12, 14, 16]. Similarly to our approach, this can lead to random transient errors appearing during operation. Alternatively, the designs of the circuits themselves can be altered to incorporate approximations that reduce the critical path length [20, 21, 23, 26, 43], which leads to circuits that always produce errors for specific inputs. Both approaches look into taking advantage of known CMOS properties, while our work focuses on emerging non-CMOS technologies with intrinsic multi-level support. Extensive work also exists in using approximate computing to reduce the costs associated with the memory system, both for single [13, 24, 32] and multi-level [41] memories. Jain et al. [19] propose using smart floating point width reduction when storing data in the cache as a compression method, while still utilizing full width precision when performing calculations. This is an intriguing idea that could be used in parallel with our work. Miguel et al. propose combining load-value prediction and approximate computing in their Load Value Approximation work [29], mitigating the costs of the memory system in the core instead of the memory itself. Finally, approximate CPUs have been proposed [12, 48], but once again, they are based on CMOS technology and neither of them propose any changes that affect the cache storage capacity.

Previous work [6, 8, 17, 22, 34, 37, 41, 42, 42, 51] has shown that it is possible to develop multi-level devices capable of replacing CMOS, but the potential on a full system has not been evaluated. This inspired our initial work on multi-level technologies [44, 45], which, in turn, is the basis for this work. In our previous work, we introduced some initial designs for parallel adders and caches and evaluated them using the EnerJ [40] framework. In this work, we modify the adder to a simpler design that utilizes fewer gates, introduce a multiplier and a shifter, and evaluate all of them in more detail using SystemC [**?**]. We also use a more detailed architectural simulator, Gem5 [7] instead of EnerJ, which makes it possible to more accurately estimate the energy and performance characteristics of the circuits when used in a general-purpose CPU executing some commonly found applications. Overall, in addition to the new logic circuits, the evaluation methodology has been redesigned from the ground up, both to enhance accuracy and to apply it to more realistic scenarios.

On the software side, various frameworks for evaluating the effects of approximate computing exist [9, 30, 39, 40, 47, 50]. While there are similarities to the framework we have developed, none of them fully fulfilled the requirements for our evaluation. For example, iACT [30] utilizes Pin [28], much like us, but it does not model the same approximations and errors as we do. ACCEPT [39] focuses only on software based approximations, but we still use the benchmarks and the compiler it provides. EnerJ is implemented for Java applications and cannot simulate the runtime performance in as much detail as our framework. None of the tools for evaluating circuit-level approximations available to us provide sufficient support for our multi-level designs. Finally, we have not investigated software frameworks for monitoring and correcting errors during approximate computation [5, 15], but we expect that such tools can improve the range of applications our approach can support. For example, we explain in Sec. 6.2 that the `blackscholes` application cannot be used as it is with our approximate circuits, but a software solution that detects unacceptably high errors and corrects them should solve the issue. Further investigation of the potential of error correction and detection is left as future work.

3 MODULATING ERROR RESILIENCE IN EMERGING MULTI-LEVEL DEVICES

CMOS technology scaling, the current driver of Moore's law, is facing practical and fundamental limits. The search for low-power alternatives to CMOS is intensifying with a large number of emerging technologies being investigated, such as quantum point contact devices [42], single

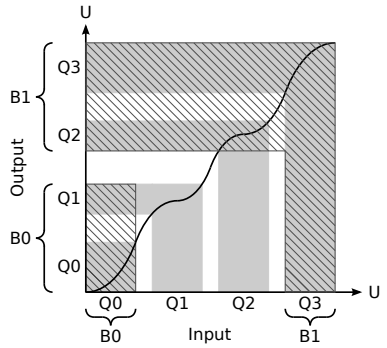


Fig. 1. Illustration of a quaternary device that can operate on binary data. Gray regions represent the input and output ranges when working with quaternary data. Striped regions represent input and output ranges when working with binary data. The figure is explained in more detail in Sec. 3.

electron [17], and carbon nanotube [34] transistors. Many such technologies present new features that provide opportunities to develop new computer systems. Multi-level devices are one such common feature [22, 37, 42] that is already employed by several commercial memory technologies, e.g., in multi-level flash [8] and phase change memory (PCM) [6, 41, 51].

In this work, we consider any device where the output can be described as a continuous transfer function of its inputs. For multi-level logic, this transfer function contains plateaus that presents a relatively stable output even if the input value varies slightly [42]. These plateaus appear as discrete ranges that can be used to represent logic states. Such devices have higher probability that the output of a device shifts, due to interference, to a nearby level than to a level further away from the expected level. Here we use the term interference for any physical property that affects the output such that it deviates from the ideal transfer function, e.g., input noise or manufacturing variances. To compensate for this issue and be able to take advantage of the multi-level capabilities of such devices, higher energy usage and longer operation times are required [17, 34, 36]. We take advantage of approximate computing to remove this requirement by creating two different logic representations for approximate and precise data.

Fig. 1 illustrates a transfer function of a fictitious quaternary device in some unit U (e.g., voltage or current). The figure shows in light gray the four different quaternary states (Q0, Q1, Q2, and Q3) and how they are represented as ranges on the input and output of the device. When representing approximate data, we use all logic states of the device. This maximizes the amount of data that can be represented per device, but it also increases the probability that a state shifts to a nearby state (e.g., Q2 gets interpreted as Q1 or Q3). Precise data is represented in binary form by only using the maximum (Q3) and minimum (Q0) states as input to the device. When the output is read nearby states are also considered as part of the maximum (Q2 and Q3) and minimum (Q0 and Q1) states (see striped regions). This increases the reliability of the device as the probability is low that Q0 would shift to Q2 or that Q3 would shift to Q1. However, it also requires a larger number of devices compared to representing the data approximately. For example, twice as many memory cells are required for storing the same amount of data precisely, and a parallel adder with twice the width (in digits) is required for adding two binary numbers together. In the next section, we describe how such multi-level devices can be used to implement complex logic circuits that can operate on either precise (binary) or approximate (quaternary) data based on the application requirements, focusing on the design choices required due to quaternary logic gates and memory cells that also need to handle binary data.

Or	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

(a) Bitwise Or

And	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

(b) Bitwise And

Xor	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

(c) Bitwise Xor

Inv	
0	3
1	2
2	1
3	0

(d) Bitwise Inversion

Min	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	2	2
3	0	1	2	3

(e) Minimum

Max	0	1	2	3
0	0	1	2	3
1	1	1	2	3
2	2	2	2	3
3	3	3	3	3

(f) Maximum

Ha	0	1	2	3
0	0	1	2	3
1	1	2	3	**0**
2	2	3	**0**	**1**
3	3	**0**	**1**	**2**

(g) Half Adder

Fa	0	1	2	3
0	1	2	3	**0**
1	2	3	**0**	**1**
2	3	**0**	**1**	**2**
3	**0**	**1**	**2**	**3**

(h) Full Adder (Carry In=1)

Fig. 2. Quaternary truth tables for some of the most commonly used gates. The shaded cells contain the results for binary data. The bold numbers in the adders indicate an output carry of "1". Note how the adders produce the wrong result for binary values.

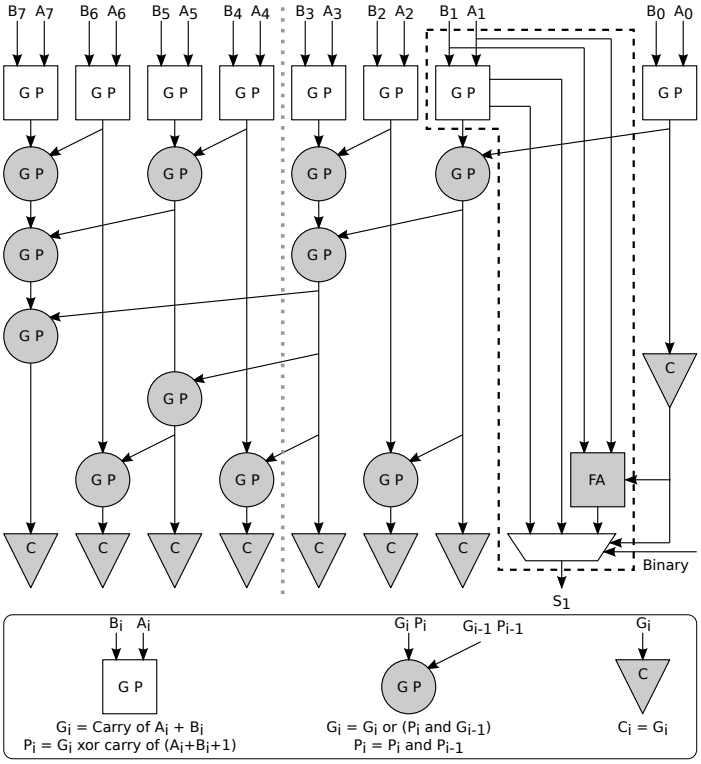
4 QUATERNARY SYSTEMS

In order to investigate the potential of quaternary circuits, we propose designs for basic arithmetic units that can efficiently work with both binary and quaternary data. These designs are based on basic quaternary logic gates that can also produce binary results (Fig. 2), as discussed by Själander et al. [44]. We also propose memories able of storing both binary and quaternary data, although not in as much detail, since memories with multi-level cells are already common in storage solutions such as SSDs. We focus on circuits that can work with both binary and quaternary data because having two separate circuits for every operation would lead to a significant increase of area and routing overhead. We should note here that we keep our designs on a relatively high level, not delving into the actual gate implementations, because our goal is to evaluate the potential of quaternary logic for circuits in general and not for some specific transistor technology. Therefore, while the exact gates used for the circuit will vary from technology to technology, the high-level design should remain the same.

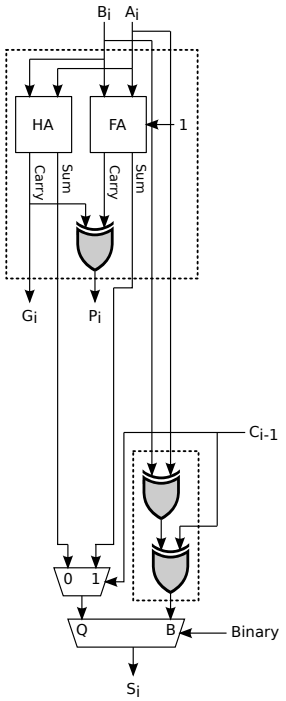
4.1 Combinational Logic Circuits

Modern systems can perform a large number of operations, but a lot of them can be expressed as a combination of simpler operations. For example, integer division can be expressed as a combination of integer subtraction and shifting. Similarly, floating point operations can be expressed as a combination of integer operations and other simpler floating point operations. For this reason, we focus on operations that either cannot be expressed as a combination of simpler operations (e.g., integer addition), or, if they can, their implementations would be significantly inefficient (e.g., integer multiplication). Specifically, we propose logic-level designs for three circuits: An integer adder, an integer multiplier, and a shifter. We also briefly discuss logical operations, but we do not go into details because they are are not as complicated to design.

4.1.1 Parallel Adder. Fig. 3a depicts a Brent-Kung carry-lookahead adder that has been adapted to support both binary and quaternary operations efficiently. Gates in white operate with both quaternary and binary data, while gates in gray only operate with binary. A similar design for a



(a) Overview of the adder circuit. The part of the figure indicated with the dashed outline is depicted in more detail in Fig. 3b.

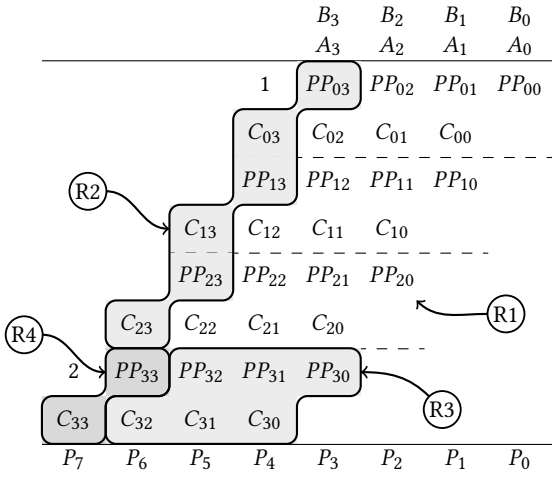


(b) The modified initial Generate-Propagate and final addition modules for the quaternary adder.

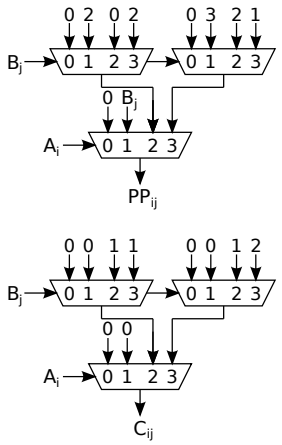
Fig. 3. The Brent-Kung adder that can operate on both binary and quaternary data. The white gates are quaternary gates, while the gray gates are binary only.

Kogge-Stone adder is described in our previous work [44], so here we will only provide an overview. When the adder is operated in quaternary mode, only the right half (separated by the gray dotted line in the figure) is fed with data. Because the quaternary HA and FA gates do not return the correct results for binary operations (Fig. 2g and 2h), the initial generate and propagate signal generation needs to be adapted, as shown in Fig. 3b. Similarly, the calculation of the final result needs to take into consideration if we are performing a binary or a quaternary addition and use the correct adder gates. However, the carry-lookahead tree does not need any modification and it can in fact operate using only binary gates, since adding together two one-digit numbers results in a carry value that cannot be more than "1". This has the dual advantage of both keeping the circuit design simple and reducing the probability of errors during operation. Furthermore, it is possible to perform subtraction, both in binary and in quaternary mode, by using the conventional two's complement representation of negative numbers.

Regarding the performance characteristics of the adder, we assume that the delay introduced by the additional HA and FA used to generate the initial signals is similar to the delay of the XOR gates. After all, addition is a common operation used for a lot of operations and no technology would be viable without efficient addition gates. At the same time, the final multiplexer that selects between



(a) Baugh-Wooley signed multiplication overview. The dashed lines separate the steps in the long multiplication algorithm. The shaded/encircled partial products (PPs) and carries (Cs) are the edge PPs that need to be modified for Baugh-Wooley. R1–4 indicate four different regions of PPs and Cs.



(b) Example multiplexer circuit for calculating one partial product and its carry in R1. For the other regions the values multiplexed need to be modified according to Equation 1.

Fig. 4. Eight bit, Baugh-Wooley quaternary multiplication. Unlike binary multiplication, each partial product (PP_{ij}) also produces one carry digit (C_{ij}), which needs to be added to the next column. Bit negation also differs from the binary circuit.

the binary and the quaternary result introduces negligible delay, because the control signal (binary or quaternary) is applied well before the results of the addition reach it.

4.1.2 Multiplier. While it is possible to perform multiplication by successive additions and shifts, the delay introduced by such a multiplier is significant. For this reason, we design a quaternary multiplier using the Baugh-Wooley signed multiplication algorithm (Fig. 4a) and a Dadda-like reduction tree [10]. In order to calculate all the partial products, we use a multiplier gate consisting of multiplexers (Fig. 4b), similar to the design by Moaiyeri et. al [34]. The final reduction of the partial products is performed using the quaternary adder described in Sec. 4.1.1.

Each partial product is no longer produced by multiplying individual bits, making it incorrect to implement Baugh-Wooley by simply inverting the partial products at the edges of the multiplier. On top of that, multiplying two quaternary digits produces not only a partial product digit but also a carry that needs to be added to the next partial product. However, we have determined that it is still possible to design a Baugh-Wooley multiplier for quaternary data.

We use the notation $(D_{N-1}D_{N-2}...D_0)_B$ to represent an N-digit number in base B. For example, $(Q_n)_4 = (b_1b_0)_2$ indicates that the quaternary digit Q_n consists of the two binary digits (bits) b_1 and b_0, with b_0 being the least significant bit. With this notation, for $A \cdot B = P$ and for each partial

product $A_i \cdot B_j$, we have:

$$\begin{aligned}(A_i)_4 \cdot (B_j)_4 = (C_{ij}PP_{ij})_4 &= (a_{i1}a_{i0})_2 \cdot (b_{i1}b_{i0})_2 \\ &= (a_{i0} \wedge b_{j0}) \\ &+ ((a_{i1} \wedge b_{j0}) \oplus r(j)) \cdot 2 \\ &+ ((a_{i0} \wedge b_{j1}) \oplus r(i)) \cdot 2 \\ &+ ((a_{i1} \wedge b_{j1}) \oplus r(j) \oplus r(i)) \cdot 4 \qquad (1)\end{aligned}$$

$$(P_{MSB})_4 = (\overline{p_{MSB,1}}p_{MSB,0})_2 \qquad (2)$$

with

$$r(n) = \begin{cases} 0, & \text{for } n \neq \text{width} - 1 \\ 1, & \text{for } n = \text{width} - 1 \end{cases}$$

Equation 1 depicts how the quaternary digit multiplication needs to be adapted to support signed Baugh-Wooley multiplication. Expanding it for each region R in Fig. 4a, produces:

$$A_i \cdot B_j = \begin{cases} (a_{i0} \wedge b_{j0}) + ((a_{i1} \wedge b_{j0}) + (a_{i0} \wedge b_{j1})) \cdot 2 + (a_{i1} \wedge b_{j1}) \cdot 4, & for\ (R1) \\ (a_{i0} \wedge b_{j0}) + ((a_{i1} \wedge b_{j0}) + \overline{(a_{i0} \wedge b_{j1})}) \cdot 2 + \overline{(a_{i1} \wedge b_{j1})} \cdot 4, & for\ (R2) \\ (a_{i0} \wedge b_{j0}) + (\overline{(a_{i1} \wedge b_{j0})} + (a_{i0} \wedge b_{j1})) \cdot 2 + \overline{(a_{i1} \wedge b_{j1})} \cdot 4, & for\ (R3) \\ (a_{i0} \wedge b_{j0}) + (\overline{(a_{i1} \wedge b_{j0})} + \overline{(a_{i0} \wedge b_{j1})}) \cdot 2 + (a_{i1} \wedge b_{j1}) \cdot 4, & for\ (R4) \end{cases}$$

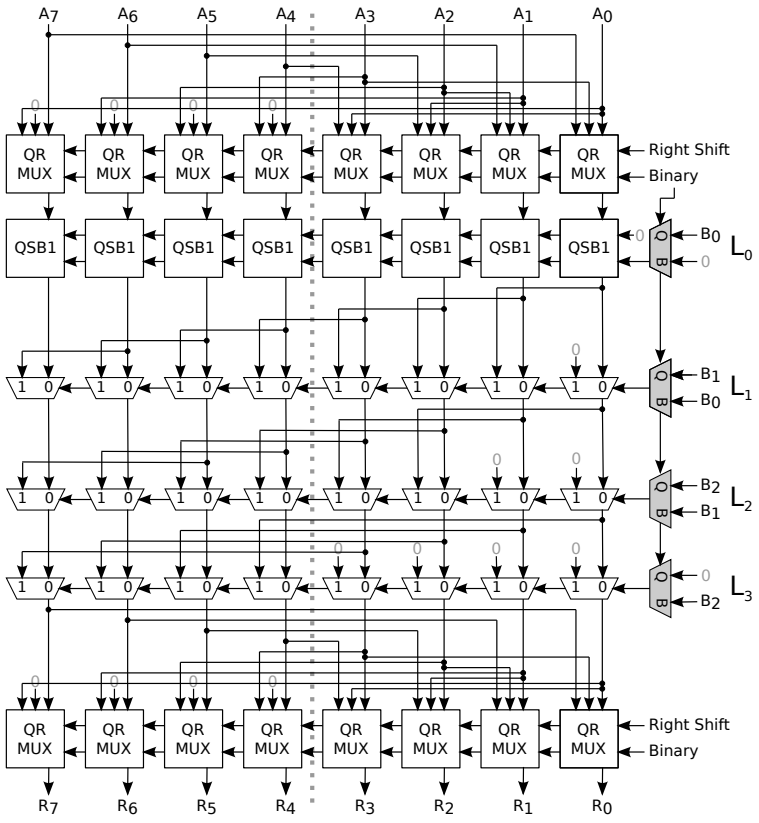
After the partial products have been reduced, the result needs to be corrected by inverting its most significant bit, as seen in Equation 2. This is achieved at the circuit level by introducing an additional partial product with a constant value of "2", as adding that into a quaternary digit causes its most significant bit to invert. Since we cannot know if there is some more efficient way of implementing the partial product modules in the underlying quaternary technology, we assume the worst case scenario and model them as a combination of multiplexer gates, with an example for region R1 depicted in Fig. 4b.

Similar to addition, quaternary multiplication does not produce the correct results for binary data. However, unlike addition, we would have to further adapt and add additional gates in all levels of the multiplier, not just at the beginning and end. Since this would most likely require multiplexers to choose between the binary and the quaternary results, the total area overhead would exceed that of a second multiplier. For this reason, we decided to split the binary and the quaternary circuits into two separate circuits that share only the final adder.

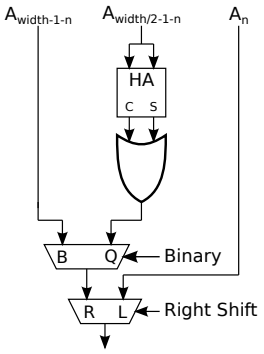
Another major difference between the adder and the multiplier is that signed multiplication requires a different circuit than unsigned. In order to support both, the left-most and the bottom row partial product gates need to be modified in order to invert all or some of their output bits, much like in a conventional binary Baugh-Wooley multiplier.

Finally, we also considered using a Booth multiplier instead of Baugh-Wooley. When implementing Radix-4 Booth multiplication with recoding, it is necessary to shift the partial products by one bit, which cannot be achieved trivially when working with quaternary values. The reason why is further discussed in Sec. 4.1.3.

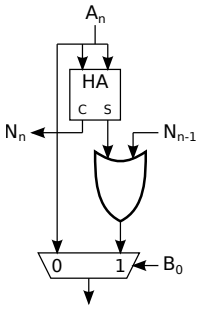
4.1.3 Shifter. We propose a quaternary barrel shifter design that works with both binary and quaternary values (Fig. 5a). Binary barrel shifters are implemented as levels of multiplexers controlled directly by the shift-by input's bits, each shifting the input by a power of two [18]. In order to



(a) Overview of the shifter.



(b) The Quaternary Reversal Multiplexer (QR MUX) module.



(c) The Quaternary Shift-By-One (QSB1) module.

Fig. 5. A quaternary bidirectional shifter capable of operating with both quaternary and binary data. The gates shaded gray are binary gates, while the rest are quaternary.

support both left and right shifts efficiently, two data reversal levels are added before and after the shift multiplexers, and are responsible for reversing the direction of the input bits. Since reversing the bits in a quaternary digit is not as trivial as multiplexing between two signals, we have designed a custom quaternary reversal multiplexer (QR MUX), depicted in Fig. 5b. In order to make sure that during quaternary operations the data will not be reversed into the left (inactive) part of the circuit, the reversal multiplexers have three inputs instead of two, creating two different pivot points for reversing binary or quaternary data. With this design, it is possible to shift by an arbitrary amount with only $\log_{BASE} INPUT\_WIDTH$ levels of multiplexers.

When using this design with quaternary gates, shifting by one digit results into a shift by two bits. For example, L_1 in Fig. 5 shifts its input by one bit when used with binary data and by two bits (one quaternary digit) when used with quaternary data. Therefore, it is possible to use the same design as for the binary shifter, as long as we have a special level of quaternary "Shift-By-One" (QSB1) modules for shifting quaternary digits by one bit, indicated in the figure as L_0. Fig. 5c shows how this can be achieved using HA gates, but depending on the underlying technology, a more efficient implementation might be possible. The most significant bit of the quaternary digit is extracted and forwarded to the next module. While this looks similar to how the carry chain of an adder works,

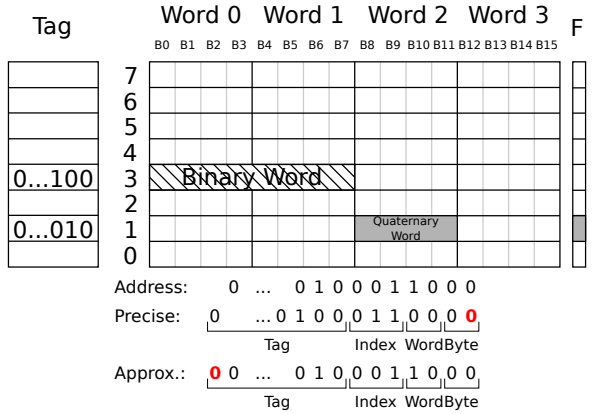


Fig. 6. A direct mapped cache that can store both binary and quaternary data. The shaded cells represent the word that the address points to. While the figure depicts a direct mapped cache, additional ways can be added without any design modifications.

it only introduces one additional level of delay, because the "carry" digit (N_n) can be calculated using only the input from the corresponding reversal module and does not require any data from the previous QSB1 module. By implementing the original shifter design with quaternary gates, and adding this special quaternary shift-by-one level, it is possible to operate on both binary and quaternary data simply by multiplexing the shift-by input bits to the correct level. When operating with quaternary data, the bits of the shift-by input are fed into levels L_0 to L_2, while binary data are processed by levels L_1 to L_3 instead. In order to reduce the complexity of the circuit, the shift-by amount is always given in binary, even if the data are originally stored in quaternary format. This can be easily done using only one level of gates, so it does not introduce any significant overheads.

4.1.4 Logical. Logical operations can operate either bitwise (e.g., Xor) or on the whole data (e.g., integer comparison). It is clear from Fig. 2a, 2b, 2c, and 2d that the quaternary gates are capable of correctly working with binary data as well. At the same time, comparison circuits are fairly simple and depend only on simple bit comparison operations, so no special consideration is required for these as well. Given that logical circuits are trivial to implement, we do not present a design for them, nor do we discuss them in detail.

4.1.5 Converting between binary and quaternary. In order to perform the operations described, the data needs to be in the correct representation. Better detailed in Sec. 3, this means that binary data should use the two most extreme levels in the device, with the levels used for the binary "0" and "1" corresponding to quaternary "0" and "3" respectively. Additionally, when converting from binary to quaternary, the binary bits need to be packed to quaternary digits in pairs, while during the reverse operation quaternary digits needs to be unpacked into two binary bits. All of these operations can be achieved using simplified DAC and ADC circuits, for which solutions already exist in production thanks to the widespread usage of Multi-level Cell (MLC) flash memories. It is also possible that multi-level circuits will have specialized solutions based on the characteristics of the underlying technology.

4.2 Memory

Multi-level technologies are already used in memory systems [36], but the data are always stored without any errors, at the cost of increased energy and time requirements.

In our previous work [44, 45], we have proposed a cache design that can store both binary and quaternary data (Fig. 6), without the need to have a statically defined split between binary and quaternary cache lines. For this, we assume that data are classified as either precise (binary) or approximate (quaternary) when they are inserted into the cache and remain so until they are evicted. We will briefly discuss this design here, and also discuss the challenges and possibilities when applying similar techniques in the main memory and the register file.

The main idea behind the proposed design is that we can treat memory cells as either binary or quaternary dynamically, affecting their capacity and error resiliency. A cache line consisting of 512 multi-level cells is able to store 128 bytes of data in quaternary mode, but only 64 bytes in binary mode. Essentially, the cache capacity changes based on the type of data stored in the cache.

There are multiple ways of taking advantage of this, such as treating storing data in quaternary format as cache compression [2, 4, 11], or treating quaternary cache lines as two separate cache lines, increasing the associativity of the set containing them. However, both of these ways greatly increase the cache complexity, introducing unnecessary overhead that needs to be paid even when using only precise (binary) data. For this reason, we propose an alternative, simpler, solution: The cache is designed as a normal cache, but quaternary cache lines are simply treated as being twice as large as binary cache lines. In addition to increasing the available cache capacity for quaternary data, the energy required for accessing binary data in the cache can be reduced with memory banking techniques. To account for the fact that a binary cache line holds data spanning a smaller address range, the address is shifted by one to the left when calculating the line, index, and tag values. A single "approximate/precise" bit flag per cache line is then enough to be able to reconstruct the address when accessing the line.

This approach works for the cache because different addresses mapping to the same cache line do not affect the correctness of the cached data. On the other hand, for main memory, we need a one-to-one mapping between addresses and physical cells, so this design is not applicable. For this evaluation, we are assuming that the main memory is statically split into a binary and quaternary region, each with its own part of the address space. Assuming a quaternary memory bus as well, each request is able to carry twice as much data, meaning that the memory system does not incur any additional overhead due to the larger cache lines. Previous work by Qureshi et al. [36] has demonstrated a more advanced system for dynamically partitioning the memory.

Finally, it is also possible to use quaternary registers. One approach would be to have a set of binary registers, each of which can also be addressed as two quaternary ones. Another approach would be to simply allow storing either binary or quaternary data in all the registers, but only half of the register is used in the quaternary case. While the latter may cause parts of the register file to be wasted, it is simpler to support in hardware and in a compiler.

4.3 Instruction Set Architecture

In the designs proposed in Sec. 4.1, we only describe how the circuits can be implemented, and not how these can be integrated into a modern CPU. We consider a detailed evaluation of different instruction set architectures (ISA) to be orthogonal to our designs and outside the scope of this paper. Different ISA options are, therefore, only briefly discussed. This issue can be approached at different granularities, from a per instruction or per memory address classification to larger region based classifications. For the rest of this discussion, we assume that a very fine granularity is required, both because it offers the highest amount of control on what is approximate, and because it is possible to emulate coarse granularity region-based classification schemes using a finer granularity scheme.

We start by separating the different schemes into two large categories: Data and instruction based schemes. In the former, data is classified as either approximate or precise; instructions are then

executed approximately or precisely depending on the data they are executed with. This scheme has the advantage that, depending on the granularity, it can be implemented with a small overhead. For example, one could split the virtual memory address space into large regions, with each region storing exclusively approximate or precise data. The physical memory is then dynamically adjusted based on the type of the data stored. Exactly how this can be achieved efficiently at runtime is beyond the scope of this work, but similar approaches have been investigated before [27, 36]. By doing so it is possible to differentiate between approximate and precise data based only on their physical address. When that address is not readily available, such as when the data are moved into a cache or a register file, an additional flag bit can be used to classify the data with a small storage overhead. Then, instructions operating on the data only need to check that flag to determine if they should be executed approximately. With this approach it is possible to use approximate memories in the whole memory hierarchy with low overhead, as data always remain either approximate or precise. Reclassifying the data at runtime is possible, but it requires moving the data into a different region, which can be costly if done often or for large amounts of data.

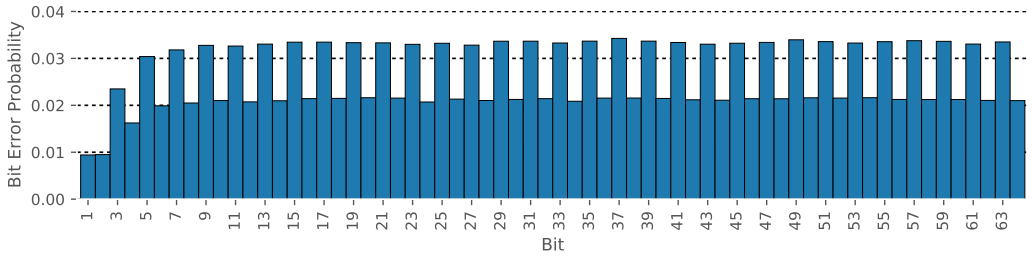
A different approach would be to instead differentiate between approximate and precise instructions. With this classification it is possible to have data that are treated both as precise and as approximate, depending on the instruction using them. This can be useful as previous research [31] has shown that applications often have phases that are more tolerant to approximation than the rest of the execution, and with this scheme the cost of reclassifying data (due to the data being converted and rewritten into the memory) can be avoided.

For this classification method, the simplest approach would be to introduce different instructions for each approximate operation. This of course would lead to a large increase in the instruction set supported by the CPU, which in turn would lead to significant bloat, both on the compiler and the CPU side. Instead, we could use one of the existing methods employed in modern hardware for controlling the runtime behavior of instructions, such as special registers that can be set programmatically, instruction prefixes or suffixes, and mode setting instructions. It is also possible to design a system where the instruction stream is not altered, but instead the addresses (PC) of the approximate instructions are fed separately to the CPU as metadata. This has the advantage that it does not increase the size of the instruction stream, which could have lead to severe performance degradation. While this requires separate data to be fetched and cached, this data are not as performance critical as the instruction stream, since in the case of missing information the instructions can safely default to precise execution. During our investigation, we did not encounter any executables containing more than a few hundred approximate instructions, which indicates that a small metadata cache (<1KiB) can be enough.

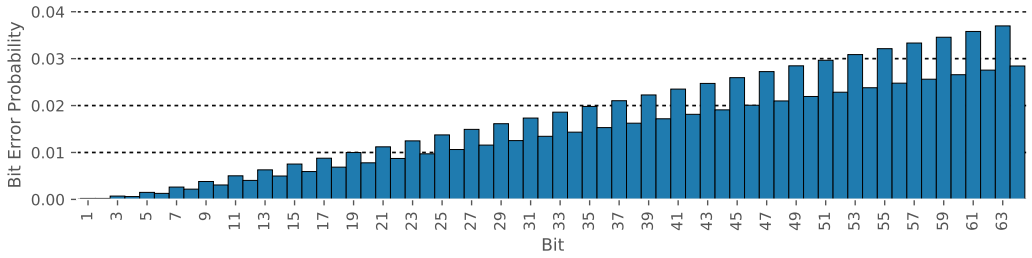
For our evaluation, we chose a combination of both approaches, explicitly classifying both data and instructions as approximate, as this approach offers a lot of flexibility when evaluating different approximations. Data are classified by their physical address, while instructions use the metadata approach.

5 EVALUATION OF LOGIC CIRCUIT OUTPUT ERRORS AND ENERGY USAGE

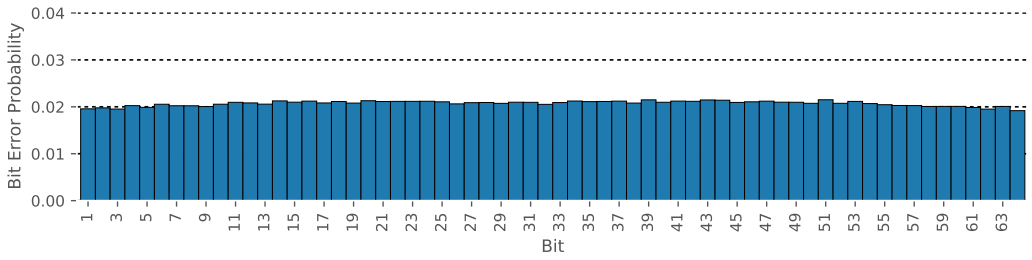
We model the three circuits described in Sec. 4.1 on a gate-level granularity using SystemC [**?**]. We chose SystemC because it enables us to model the logic gates in enough detail to get meaningful, relative error and energy numbers without forcing us to assume a specific underlying technology. Each gate is modeled as a SystemC module, with input and output ports. At each cycle, the quaternary gates have a chance of producing a transient erroneous result, based on a user-defined **Gate Error Probability**. Since we are not simulating a specific technology with well defined error probabilities, we use an arbitrary range that offers the highest amount of variability during execution. Specifically, when we later evaluate the behavior of benchmarks running on the proposed



(a) Brent-Kung Addition



(b) Baugh-Wooley Multiplication



(c) Bidirectional Barrel-Shifter

Fig. 7. Bit Error Probabilities (per bit, once an error has occurred) for integer addition, multiplication, and shifting. The operations are performed in quaternary mode, with each quaternary digit being depicted as two bits.

circuits, we can induce between hundreds of thousands to only a couple of errors in the applications' execution by varying the probability between 10^{-5} and 10^{-10}. As we cannot know how a gate's complexity will vary under each quaternary technology, we model all the different logic gates to have the same gate error probability. With this setup, we perform billions of simulations with random inputs and collect the distribution of the individual output bit errors (**Bit Error Probability**, Fig. 7), the total probability of an error appearing in the output of a circuit (**Output Error Probability**, Fig. 8), as well as the average energy used (Fig. 9). The Output Error Probability gives us the chance of getting an error, while the Bit Error Probability gives us the behavior of the circuits when an error occurs. Based on these numbers, we estimate the energy and error characteristics of more complex circuits, namely integer division and floating point operations.

5.1 Bit Error Probabilities

Fig. 7 presents the bit error characteristics for the circuits that we have modeled in detail using SystemC. We use a Gate Error Probability of 10^{-7} because it is low enough that the specific characteristics of each circuit are not drowned by excessive amounts of errors. For higher probabilities, the number of errors per calculation is so high that the results start approaching uniform distributions. On the other hand, for lower probabilities, the shape of the distribution remains the same. We present the individual bit errors instead of a metric such as the Mean Square Error (MSE) because the MSE does not provide any insights in the actual behavior of the circuits. In the aforementioned bit error figures, the individual bit error probabilities are calculated by dividing the number of times each bit is wrong by the total number of times that the output has been wrong. In other words, we measure, given that the circuit output is wrong, what is the probability of each individual output bit being wrong.

Fig. 7a contains the bit error probabilities for the adder circuit. We observe that, with the exception of the first two quaternary digits (four least significant bits, as each quaternary digit contains two bits), the error probability per quaternary digit is uniform. More interesting, however, is the fact that the probability of an error appearing in the first bit of a quaternary digit (e.g. bit 3 in Fig. 7a) is significantly higher than that of an error appearing in the second bit (in this example, bit 4). This can be explained by the fact that when a wrong carry value is produced, it has a higher chance of affecting the first bit of the next quaternary digit rather than the second. The carry can be either "0" or "1", and adding 1 to a binary number always inverts the least significant bit.

We observe a similar pattern in the multiplier circuit (Fig. 7b). The difference between the multiplier and the adder is that the error probability steadily increases as we move from the less to the more significant bits, which can be explained by the fact that the reduction tree for the multiplier gradually increases in height.

Finally, regarding the barrel shifter, the bit error probabilities are uniformly distributed across the bits. This is not surprising, as each bit is independent from the other bits.

5.2 Estimating Error Probabilities for More Complex Logic Circuits

More complex logic circuits, such as integer division and floating point operations [46], can be built using the basic circuits we have described. For example, integer division can be implemented using the long division algorithm, which works by iteratively shifting the remainder and subtracting the divisor from it. Similarly, floating point addition can be implemented using shifts to normalize both numbers to the same base and then performing integer addition on the mantissas. The same principle applies to other, even more complicated operations, such as floating point multiplication and division.

Coming up with circuit designs for all these operations is beyond the scope of this work, as the most interesting parts have already been discussed in the previous sections. However, our goal is to evaluate the effects of quaternary approximate circuits on benchmarks executing on a general-purpose CPU, so we still need some error probabilities to use during the evaluation. For this reason, we decided to estimate these probabilities based on the probabilities of the basic circuits comprising the more complex ones. In order to keep the problem tractable, and since we are only interested in roughly estimating the various probabilities, we do not consider the conditional probabilities between the output bits, as well as the conditional probabilities between the operations performed in the algorithms implemented by the circuits. For example, for integer division, we estimate the probability of output errors based on the probability of an error appearing in consecutive shift and integer addition operations. As the probabilities calculated are an estimation, we do not discuss them here in detail, but note that since we assume worst case execution, the estimated

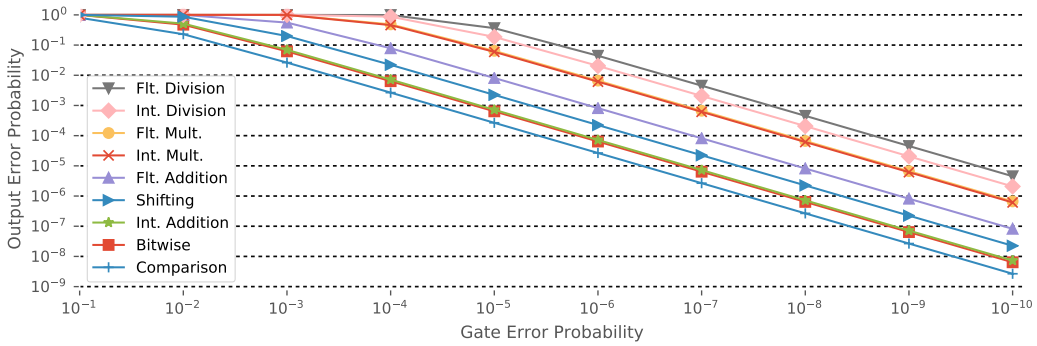


Fig. 8. The probability of an error appearing in the output of a circuit, given the probability of an error occurring in a gate.

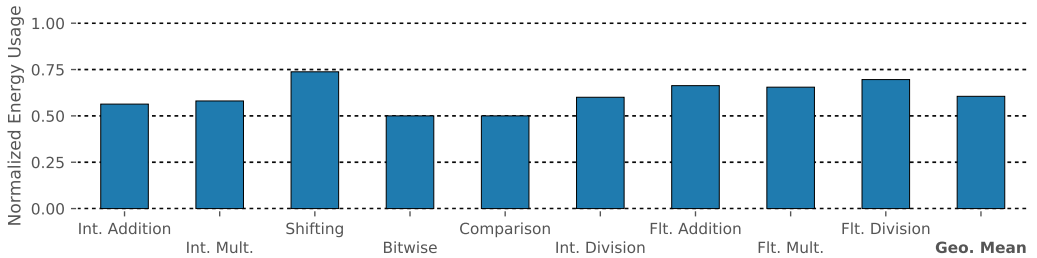


Fig. 9. Energy usage of the quaternary operations, normalized over their binary counterparts.

error probabilities are higher than they should be in practice, thus providing an upper bound for the evaluation.

Finally, in addition to the more complex operations, we also do not discuss in detail very simple circuits that are trivial to implement, such as logical and comparison circuits.

5.3 Overall Error Probabilities

Parallel to the individual bit error probabilities, we have also investigated the overall error probabilities of the circuits described in the previous sections.

Fig. 8 depicts these for gate error probabilities ranging from one out of ten to one out of ten billion. Note that both axes are in the same logarithmic scale. We observe that for low enough gate error probabilities, the output error probabilities scale linearly. We also observe that this linear scaling begins earlier for the less complex circuits. This is expected, since the fewer gates a circuit consists of the lower the probability of getting more than one gate error per cycle is.

We can conclude, from all the different circuits, that the error probabilities vary significantly, both generally and when examining individual bit errors. For example, division circuits have output error probabilities that are several orders of magnitude higher than that of addition circuits. This will have to be taken into consideration when designing a system with specific error margins, for example by using gates that are less error prone for the division circuits.

5.4 Energy Usage

The multi-level behavior shown in Fig. 1 naturally appears in emerging devices such as single electron (SET) and single atom transistors (SAT). The inherent multi-level characteristic of these devices enables the design of a multi-level half adder consisting of only two devices, one for calculating the sum and one for calculating the carry [35]. Given that complex multi-level operations can be performed on a single device we assume equal area for binary and quaternary implementations of the same operation. We also assume that the supply range of binary devices is such that reliable operations can be performed and that quaternary devices operate with the same maximum supply range, thus, achieving the same reliability when operating in the binary mode. For simplicity, we assume that a gate dissipates equal amount of energy both in binary and quaternary mode even though the input/output-swing for the quaternary mode is reduced. Thus, we use gate equivalents to estimate the area and energy of the proposed circuits.

Fig. 9 contains the energy usage of the quaternary operations, normalized over the energy usage of performing the same operation in binary. We calculate the energy usage based on the average number of gate activations happening during each operation.

For this evaluation, we did not assume that it is possible to efficiently gate the inactive part of the circuit, so it is possible to have some operations that affect that part as well. For example, if a quaternary addition overflows, the additional carry activates some of the gates in the half of the adder that is not normally used during quaternary operations. In practice, this means that our results are pessimistic and, depending on the capabilities of the underlying technology, it is possible to achieve even lower energy usage.

With this in mind, we observe that the relative energy usage for the quaternary adder is at 56%, which is higher than the number we would ideally expect based on the number of gates that are needed for the operations (ca. 50%). Similarly, the barrel shifter also uses significantly more than half the energy (74%), due to the fact that during quaternary operation the multiplexers in the inactive part of the circuit are still activated. Furthermore, the energy usage of the integer multiplier (58%) also exceeds the ideal number, but in this case the major culprit is the fact that the partial product modules are more complex in the quaternary part of the circuit. Since, as we mention in the description of the circuit (Sec. 4.1.2), we cannot know if the underlying technology allows for a quaternary partial product gate implementation that is as energy efficient as its binary counterpart, we have to assume that each quaternary partial product requires more energy to be calculated.

For bitwise and comparison operations, as well as the more complicated circuits, we again estimate the potential energy usage based on that of the basic circuits. The bitwise and comparison operations require half the energy of their binary counterparts, since the circuits remain the same but are now half in width. For the more complex circuits, the results are bound by the energy usage of the basic operations that comprise them.

Overall, we observe that the quaternary operations, can achieve a mean energy usage of 60%, when compared to the equivalent binary operations. Depending on the capabilities of the underlying quaternary technology, such as being able to efficiently gate parts of the circuit, it is possible to achieve even further energy usage reductions.

6 EVALUATION OF POTENTIAL APPLICATIONS

To determine the viability of the approximate designs discussed in the previous sections, we evaluate them using two image processing applications, namely `jpeg` and `sobel` from the ACCEPT benchmarks [39], as well as the `blackscholes` financial modeling application. We chose the first two applications because together they cover a large range of approximate operations, with `jpeg` using integer addition, multiplication, and shifting (Table 1), and `sobel` using various floating

Table 1. The runtime instruction mix for jpeg, sobel, and blackscholes.

	jpeg				sobel				blackscholes			
	Approximate		Precise		Approximate		Precise		Approximate		Precise	
Int. Addition	1638528	(47%)	1851560	(53%)	1300500	(29%)	3127407	(71%)	0		24431290	
Int. Division	0		2		0		1		0		1	
Int. Multiplication	1572864	(100%)	129	(0%)	0		0		0		1	
Shift	721024	(80%)	184470	(20%)	0		5		0		11981248	
Logical	0		699817		1560601	(43%)	2086594	(57%)	3932160	(10%)	36951849	(90%)
Flt. Addition	0		0		2080800	(53%)	1820701	(47%)	11013611	(18%)	50052293	(82%)
Flt. Conversion	0		0		1820700	(54%)	1560601	(46%)	17039360	(74%)	5894656	(26%)
Flt. Division	0		0		260100		0		2621440	(100%)	3011	(0%)
Flt. Multiplication	0		0		260100		0		20971520	(35%)	39393594	(65%)
Other	0		5104094		0		7562301		0		217335333	
Total	3932416	(33%)	7840072	(67%)	7282801	(31%)	16157610	(69%)	55578091	(13%)	386043276	(87%)

point operations. Blackscholes was chosen because, unlike the two other applications, each individual value of its output needs to be reasonably correct for the output to be usable. Since between these three applications we have millions of approximate instructions of different types, they are enough to fully exercise all the approximate circuits we have proposed. For jpeg and sobel, roughly one third of the total number of instructions executed at runtime are executed approximately, while for blackscholes less than one out of six instructions are approximate. Finally, all three applications produce outputs that can be easily presented and understood, which was another reason for selecting these benchmarks. We use the familiar "Lenna" input (512×512px, in grayscale) for jpeg and sobel, and the "64K" input for blackscholes. We were able to simulate the majority of the benchmarks available in ACCEPT, both the PARSEC and the PERFECT ones, but since the focus of this work is on the circuits and not which specific applications are amenable to approximation, we decided to not include them in the results.

We decided to perform the evaluation by simulating a conventional general purpose CPU because we are interested in evaluating the potential of quaternary technologies for replacing the conventional binary CMOS systems used nowadays. CPUs require complex control logic that takes up a significant part of their total energy usage, limiting the potential benefits of performing computations approximately. Therefore, general-purpose CPUs are a harder target for approximation, than e.g., accelerators, where most of the energy is used for moving data and performing computations.

Previous work [38, 47] has developed frameworks for tracking which variables and errors affect the application, but not on the level of tracking individual random bit errors and correlating them with the final output. The computational resources required for such a task would be significantly greater, making the approach impractical when a large number of program executions is required. At the same time, the resulting relations of such an analysis can be too complicated to give meaningful insights. For these reasons, we instead discuss the output of the applications based on the algorithm implemented and the instruction mixes found in Table 1.

6.1 Methodology

Fig. 10 outlines the process we use for evaluating the applications. The whole process is part of the framework we have developed using existing tools and adapting them for our purposes.

The first part of the process involves identifying which instructions and data in the application can be treated as approximate. This is the responsibility of the programmer, as expert knowledge in the algorithm and the output error tolerance is required, but tools to help that process exist [38, 47]. For our evaluation, we use benchmarks from the ACCEPT suite, which come annotated with which data can be approximate and which not. These annotations are fed into the clang-based

Table 2. The simulation parameters used for the evaluation.

Parameter	Value
Processor type	In-order x86-64 CPU
Issue width	2
Cache block size	64/128 bytes
L1 cache	16KiB, 2-way, 2 cycles access time
L2 cache	64KiB, 8-way, 20 cycles access time

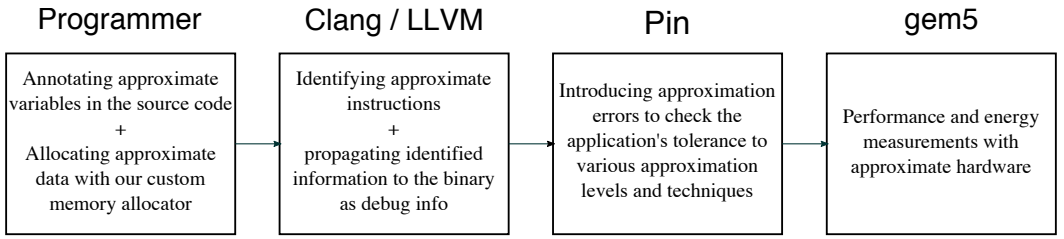
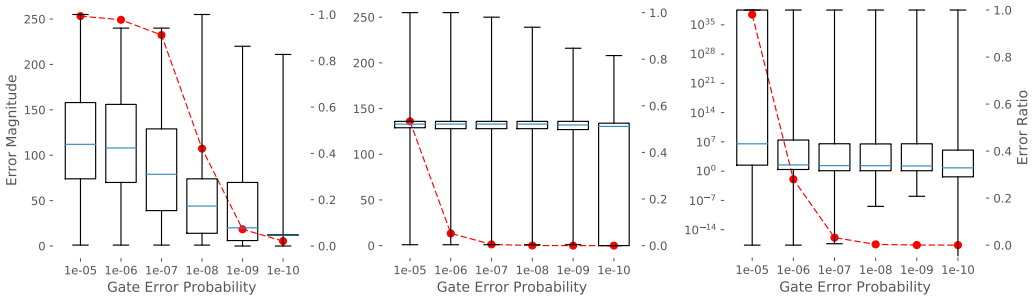


Fig. 10. The process used for evaluating the proposed quaternary hardware with benchmark applications.

`enerclang` [39] compiler, which propagates them to all the expressions using the approximate data, while also making sure that the programmer has not accidentally mixed in precise computations.

Up to this point, we have used only what the ACCEPT framework provides, with just some minor modifications. However, the ACCEPT framework evaluates approximate computing on a higher level than we do, using compiler optimizations to introduce approximations. For our evaluation, we need to be able to execute binaries on modified (simulated) hardware, so we have to transfer the metadata from the compiler to the binary. To do this, we decided to take advantage of the support for debug information available. We developed a custom LLVM pass that, based on the metadata available to the compiler, adds custom debug information to all of the approximate instructions. After the final binary for the application has been created, we can extract the embedded debug information for it and easily identify which instructions are approximate and which not. In addition, we modified the applications to use a custom memory allocator for allocating approximate data, which can be parameterized to use separate address ranges for approximate and for precise data. One advantage of this approach is that the binaries generated can also be executed natively, as we have not modified anything other than the debug information.

With the approximate instructions and data identified, we can then run the binaries in Pin [28], a dynamic binary instrumentation tool developed by Intel. It works by modifying the binaries on the fly using a JIT engine, and then executing them natively, which is why it is important for us that the binary can be executed natively. How a binary is instrumented and what the instrumentation does is controlled by dynamic libraries called "pintools". For the purposes of this work, we developed our own custom pintool that uses the debug information available in the binary to identify which instructions are approximate and then introduces errors to their results. Specifically, every time an approximate instruction is executed, the pintool identifies the output register or memory location of the instructions and then lets the instruction execute unhindered. After the instruction has finished executing, the pintool consults the tables containing the output error probability for the operation (Sec. 5) and determines if an error should be introduced or not. If it determines that it should, then it consults the bit error probabilities for the operation to determine which bits should the error affect. Finally, the output value of the operation is modified to incorporate the error. A similar approach is



(a) jpeg (b) sobel (c) blackscholes

Fig. 11. Error magnitude (boxplots – left axis) and error ratio (lines – right axis) for the benchmark applications.

used to introduce errors in the memory system, but by instrumenting all instructions that perform memory loads instead. With Pin, we then execute the benchmarks hundreds of times, collecting the amount of errors and the outputs they produce. Pin has the advantage that it introduces relatively low overhead to the execution of the application, especially when compared with the next tool, Gem5. For example, for `jpeg`, the execution time in Gem5 is 140× slower than in Pin[1], which is why we use Pin for evaluating the tolerance of the applications to errors.

However, Pin cannot give us full system performance or energy numbers. For these we instead use the Gem5 detailed, full system simulator [7]. Gem5 enables us to simulate a conventional CPU and memory system, while maintaining detailed statistics about every single aspect of the execution. Like Pin, it can run unmodified binaries, so we use the same binary for both Pin and Gem5. As we have already mentioned, Gem5 is significantly slower than Pin, which makes running thousands of benchmarks prohibitively expensive in computational resources. For this reason, we use Pin for the initial output error evaluation and then we use Gem5 for the more detailed performance and energy numbers. In order to do this, we had to modify Gem5 to be able to differentiate between approximate and precise instructions and data, and to add support for our proposed cache design. In addition, we also employ the McPat 1.3 power modeling framework [25], which utilizes Gem5 data to calculate the energy usage of different parts of the system. Since we cannot determine the static power consumption of the underlying quaternary technology, we disregard it from this evaluation and focus on the dynamic energy only. Similarly, we do not evaluate the energy usage of the main memory, since it is not possible to determine how it affects the total system energy.

Table 2 contains some of the simulation parameters. We simulate an in-order CPU instead of a more advanced out-of-order CPU because, in the latter, the energy cost of the computation is overshadowed by the overall energy cost of the out-of-order execution engine. In-order CPUs are still widely used in energy constrained scenarios, providing a viable target for our optimizations. We assume that a quaternary memory bus is available, but we do not simulate data errors during transmission because they would only slightly increase the bit error probabilities without affecting the shape of the bit error distribution.

6.2 Applications' Tolerance to Errors

Fig. 11 depicts the characteristics of the errors in the output of the three benchmark applications. Each plot contains two separate metrics, the magnitude of each individual error in the output

[1]Simulation times vary, but generally, for `jpeg`, we observe around 20 to 30 seconds with Pin and 40 to 50 minutes with Gem5.

(boxplot) and the amount of errors relative to the total output (line plot). The magnitude is produced by iterating over all the elements in the output (i.e., pixels for `jpeg` and `sobel` and options for `blackscholes`) and calculating the absolute difference between the approximate and the correct output. Similarly, the amount of errors is the ratio of output elements that are exactly equal between the approximate and the precise outputs. We avoid using error metrics such as the MSE because they abstract away a lot of the details that provide insights in the nature of the errors. In addition to the plots, we provide sample outputs from `jpeg` and `sobel`, seen in Fig. 12 and Fig. 13, respectively.

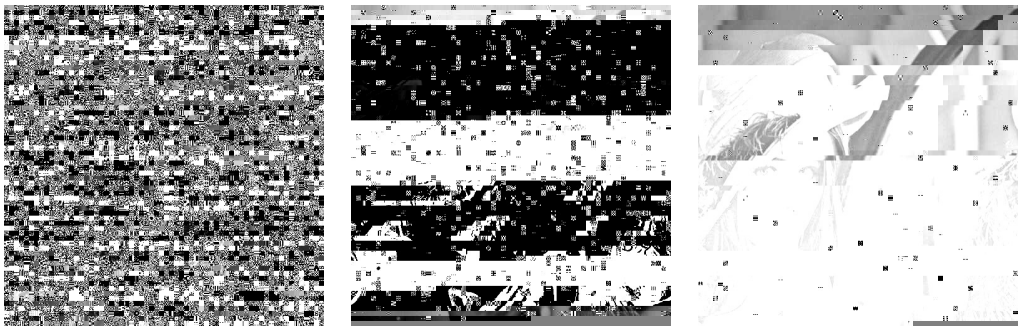
The `jpeg` benchmark implements the well known JPEG image compression algorithm. The algorithm works on blocks of 8x8 pixels and each block is differentially encoded based on the previous block. Therefore, if one of the blocks is corrupted, the rest might also contain invalid data. In Figures 12d and 12e we can see that parts of the image are shifted, in both cases starting after an error has occurred in the previous block. While this differential encoding is beneficial for the compression ratio of the algorithm, it makes the algorithm less resilient to errors. Overall, we observe that both the average error magnitude and the number of errors are reduced when the gate error probability is also reduced (Fig. 11a). However, due to the fact that bit errors can appear at any position of the output, the maximum error remains high regardless of the probability.

The next benchmark, `sobel`, implements the Sobel operator for edge detection. Unlike `jpeg`, `sobel` calculates the value of each pixel based solely on the values of its neighboring pixels, meaning that errors do not cascade and are instead isolated to the point that they occurred. For example, comparing Fig. 12c and Fig. 13c, we observe that while `sobel` faces more errors during its execution, the result is significantly better than that of `jpeg`. Similarly, if we compare Fig. 11a and Fig. 11b, the error ratio drops much quicker in `sobel` than in `jpeg`. Another interesting observation is that the median output error remains centered in the plot regardless of the error probability. `Sobel` uses floating point operations, which tend to have large errors, but the results are then clamped to eight bit integers. This leads to a lot of the errors manifesting as pixels with extreme values (salt-and-pepper noise), particularly black. Since the majority of the output is gray, the difference between the correct pixel value and the erroneous one often lays in the middle of the output range.

Finally, the `blackscholes` benchmarks implements the Black-Scholes mathematical model for calculating the pricing of options in the financial market. This benchmark has two main differences when compared to the previous two. First, the results are floating point numbers and not integer pixel values. This means that the error magnitude can be significantly larger than before. Second, each element (option) in the output is important and needs to be reasonably precise for the output to be acceptable. This is an important difference, because in the previous applications isolated, large magnitude errors can still produce acceptable results. In Fig. 11c we observe that the magnitude of the errors can easily reach values greater than 10^{35}. Given that these numbers represent monetary value, such errors are unacceptable by any quality standard. For the output of `blackscholes` to be usable the application will have to be modified to handle such errors, for example by detecting them and repeating the necessary calculations. Such modifications are outside the scope of this work, so we will discard `blackscholes` for the remainder of the evaluation. However, `blackscholes` represents a good counter-example of an application that can benefit from approximate computing but does not work (unmodified) with our proposed designs, which is why we decided to include it as part of the applications' error tolerance evaluation.

6.3 Energy and Performance Gains

So far, we have used SystemC and Pin to establish that it is possible to treat a number of instructions and data as approximate while also maintaining a reasonable output quality. In order to determine if that number is significant enough to yield tangible energy and performance gains on a conventional hardware system, we utilize the Gem5 detailed full system simulator. From our experiments, we have



(a) Gate Error Probability = 10^{-5}
Errors: 66810

(b) Gate Error Probability = 10^{-6}
Errors: 6534

(c) Gate Error Probability = 10^{-7}
Errors: 668

(d) Gate Error Probability = 10^{-8}
Errors: 61

(e) Gate Error Probability = 10^{-9}
Errors: 6

(f) Gate Error Probability = 10^{-10}
Errors: 1

Fig. 12. Example outputs from `jpeg` with different gate error probabilities. By "Errors" we refer to the number of instructions that produced an erroneous result during execution.

observed that regardless of the error probability (within limits), the performance and energy usage of the application remain the same. Therefore, we disregard the error probability as a parameter when evaluating the applications with Gem5.

The results can be seen in Fig. 14. While our primary goal is to reduce the dynamic energy of the system, we first discuss how the approximations affect performance, since the overall energy usage of the system is tied to its runtime performance.

Fig. 14a contains different performance related metrics for the three applications we are evaluating. The numbers are normalized over the precise execution, so a value of 1.0 indicates no change when approximation is applied. Since approximate and precise instructions require the same number of cycles, only the caches affect the performance of the application. In the case of `jpeg`, both the `L1D` and the `L2` caches have a similar miss ratio between the approximate and the precise executions. This is because the application has a very small miss ratio overall, so it does not benefit from a higher cache capacity. As a matter of fact, it is possible to develop a mechanism that detect the low miss ratio and disable the use of approximation in the cache, improving the output quality without affecting the performance, but it is left as future work. On the other hand, `sobel` benefits significantly from the increased cache capacity for approximate data, with the `L1D` miss ratio dropping from 34% to 1% (97% reduction). In turn, this leads to the approximate `sobel` execution

(a) Gate Error Probability = 10^{-5}
Errors: 166744

(b) Gate Error Probability = 10^{-6}
Errors: 15325

(c) Gate Error Probability = 10^{-7}
Errors: 1525

(d) Gate Error Probability = 10^{-8}
Errors: 178

(e) Gate Error Probability = 10^{-9}
Errors: 11

(f) Gate Error Probability = 10^{-10}
Errors: 1

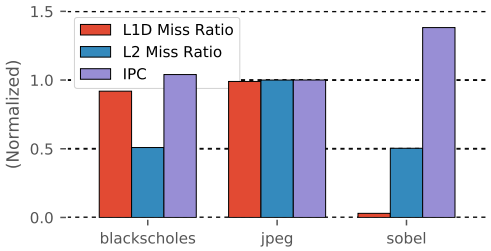
Fig. 13. Example outputs from `sobel` with different gate error probabilities. By "Errors" we refer to the number of instructions that produced an erroneous result during execution.

being faster than the precise one, with a relative IPC of 1.4. The L2 miss ratio is also decreased, but the number of accesses to the L2 are too few for it to be relevant.

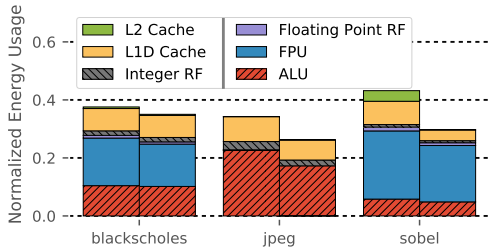
Fig. 14b contains the precise (left) and approximate (right) energy usage for different parts of the CPU, normalized over the total energy usage of the CPU during precise execution. Parts of the CPU that are not affected by the approximations applied are not presented in the figure, but are **still taken into consideration when calculating the total energy usage**, hence why the precise bars (left) do not reach the full value of 1.0.

As seen in Table 1, `jpeg` is mostly comprised of integer and logical operations, all of which are executed by the arithmetic and logic unit (ALU). Therefore, most of the energy is used by the ALU and the integer tegister file (RF). Overall, we observe a 30% reduction in the energy used by the ALU, and a 38% reduction in the energy used by the register file. In addition, we observe a 19% reduction in the energy used by the L1D cache, for a total CPU energy usage reduction of 10%.

`Sobel` benefits even more from the approximations, with energy reductions of 17% and 28% for the floating point unit (FPU) and the floating point register file respectively, as well as 54% and 98% for the L1D and L2 caches. This leads to an overall energy usage reduction of 13% for the whole CPU. Since the execution time is decreased, the static energy would have decreased with it, but we cannot evaluate this due to the characteristics of the underlying technology being unknown.



(a) Cache miss ratios, and IPC during the approximate execution, normalized over the precise execution.



(b) Dynamic energy used by select components for precise (left) and approximate (right) execution, normalized to the total energy used by the precise CPU.

Fig. 14. System Performance and Energy

Overall, both applications exhibit improved energy usage when approximations are employed, and in the case of (`sobel`) we even observe a significant performance boost. This indicates that approximate designs using quaternary technologies are indeed a viable solution, even on conventional computer systems.

These results where achieved on a general-purpose CPU where the overhead of the control logic, which needs to always be precise, is significant, both in energy and in performance. If the same designs where used on a specialized accelerator, such as an FPGA or an ASIC, the energy and performance gains would have been relatively bigger. However, as we explain in the beginning of the evaluation, our goal is to evaluate multi-level technologies as a replacement for CMOS and not just for specialized circuits.

7 CONCLUSION

In this work, we have evaluated the potential of applying approximate computing to a general-purpose CPU in the form of quaternary logic circuits. It is not possible to approximate all parts of the CPU, nor it is possible to approximate all parts of the application being executed. However, we have shown that it is possible to achieve energy (up to 13%) and performance (up to 1.4×) gains while keeping the output of the application at an acceptable quality level.

Exactly when an output can be considered to be of "an acceptable quality level" depends on the application, as well as the use case. For example, in `jpeg` we could claim that we can tolerate a few blocks being corrupted if the results are meant for computer consumption (e.g., for a neural network) but we cannot claim the same when the results are aimed for human consumption (e.g., when sharing photographs). In our experiments, we evaluated gate error probabilities that are orders of magnitude higher than those of conventional CMOS circuits used today. Therefore, even if future quaternary technologies are not able to achieve levels of reliability comparable to today's CMOS, they can still be a viable solution for applications amenable to approximation.

REFERENCES

[1] A. Alaghi and J. P. Hayes. 2013. Survey of stochastic computing. *ACM Transactions on Embedded Computing Systems* 12, 2s (2013), 92.

[2] A. R. Alameldeen and D. A. Wood. 2004. Adaptive cache compression for high-performance processors. In *Proceedings of the International Symposium on Computer Architecture*. 212–223.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 38–49.

[4] A. Arelakis and P. Stenstrom. 2014. SC2: A Statistical Compression Cache Scheme. In *Proceedings of the International Symposium on Computer Architecture*. 145–156.
[5] W. Baek and T. M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 198–209.
[6] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. M. A. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande. 2009. A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage. *IEEE Journal of Solid-State Circuits* 44 (Jan 2009), 217–227.
[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[8] D.-S. Byeon, S.-S. Lee, Y.-H. Lim, J.-S. Park, W.-K. Han, P.-S. Kwak, D.-H. Kim, D.-H. Chae, S.-H. Moon, S.-J. Lee, H.-C. Cho, J.-W. Lee, M.-S. Kim, J.-S. Yang, Y.-W. Park, D.-W. Bae, J.-D. Choi, S.-H. Hur, and K.-D. Suh. 2005. An 8 Gb multi-level NAND flash memory with 63 nm STI CMOS process technology. In *Proceedings of the IEEE International Solid-State Circuits Conference*. 46–47 Vol. 1.
[9] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of the ACM/IEEE Design Automation Conference*. 113:1–113:9.
[10] L. Dadda. 1965. Some Schemes for Parallel Adders. *Alta Frequenza* 34, 5 (May 1965), 349–356.
[11] M. Ekman and P. Stenstrom. 2005. A Robust Main-Memory Compression Scheme. In *Proceedings of the International Symposium on Computer Architecture*. 74–85.
[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 301–312.
[13] S. Ganapathy, G. Karakonstantis, A. Teman, and A. Burg. 2015. Mitigating the impact of faults in unreliable memories for error-resilient applications. In *Proceedings of the ACM/IEEE Design Automation Conference*. 1–6.
[14] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. 2006. Probabilistic Arithmetic and Energy Efficient Embedded Signal Processing. In *Proceedings of the International Conference on Compliers, Architecture and Synthesis of Embedded Systems*. 158–168.
[15] B. Grigorian and G. Reinman. 2014. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*. 248–255.
[16] R. Hegde and N. R. Shanbhag. 1999. Energy-efficient signal processing via algorithmic noise-tolerance. In *Proceedings of the of the International Symposium on Low Power Electronics and Design*. 30–35.
[17] H. Inokawa, A. Fujiwara, and Y. Takahashi. 2003. A multiple-valued logic and memory with combined single-electron and metal-oxide-semi-conductor transistors. *IEEE Transactions on Electron Devices* 50, 2 (2003), 462–470.
[18] A. Ito. 1989. Barrel shifter. (May 1989). US Patent 4,829,460.
[19] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. 2016. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 1–13.
[20] H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han. 2017. A Review, Classification, and Comparative Evaluation of Approximate Arithmetic Circuits. *ACM Journal on Emerging Technologies in Computing Systems* 13, 4 (Aug. 2017), 60:1–60:34.
[21] A. B. Kahng and S. Kang. 2012. Accuracy-configurable adder for approximate arithmetic designs. In *Proceedings of the ACM/IEEE Design Automation Conference*. 820–825.
[22] M. Klymenko and F. Remacle. 2014. Quantum dot ternary-valued full-adder: Logic synthesis by a multiobjective design optimization based on a genetic algorithm. *Journal of Applied Physics* 116, 16 (2014), 164316.
[23] P. Kulkarni, P. Gupta, and M. Ercegovac. 2011. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *Proceedings of the International Conference on VLSI Design*. 346–351.
[24] A. Kumar, J. Rabaey, and K. Ramchandran. 2009. SRAM supply voltage scaling: A reliability perspective. In *International Symposium on Quality Electronic Design*. 782–787.
[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 469–480.
[26] A. Lingamneni, C. Enz, J.-L. Nagel, K. Palem, and C. Piguet. 2011. Energy parsimonious circuit design through probabilistic pruning. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1–6.
[27] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 213–224.
[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN*

Conference on Programming Language Design and Implementation. 190–200.
[29] J. S. Miguel, M. Badr, and N. E. Jerger. 2014. Load Value Approximation. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 127–139.
[30] A. K. Mishra, R. Barik, and S. Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack*.
[31] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi. 2017. Phase-aware optimization in approximate computing. In *Proceedings of the International Symposium on Code Generation and Optimization*. 185–196.
[32] S. Mittal. 2012. A Survey of Architectural Techniques for DRAM Power Management. *International Journal of High Performance Systems Architecture* 4, 2 (Dec. 2012), 110–119.
[33] S. Mittal. 2016. A Survey of Techniques for Approximate Computing. *Comput. Surveys* 48, 4 (March 2016), 62:1–62:33.
[34] M. H. Moaiyeri, K. Navi, and O. Hashemipour. 2012. Design and Evaluation of CNFET-Based Quaternary Circuits. *Circuits, Systems, and Signal Processing* 31, 5 (Oct. 2012), 1631–1652.
[35] J. Mol, J. Van der Heijden, J. Verduijn, M. Klein, F. Remacle, and S. Rogge. 2011. Balanced ternary addition using a gated silicon nanowire. *Applied Physics Letters* 99, 26 (2011), 263109.
[36] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis. 2010. Morphable Memory System: A Robust Architecture for Exploiting Multi-level Phase Change Memories. In *Proceedings of the International Symposium on Computer Architecture*. 153–162.
[37] F. Remacle, J. Heath, and R. Levine. 2005. Electrical addressing of confined quantum systems for quasiclassical computation and finite state logic machines. *Proceedings of the National Academy of Sciences of the United States of America* 102, 16 (2005), 5653–5658.
[38] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. 2015. Monitoring and Debugging the Quality of Results in Approximate Programs. *SIGPLAN Not.* 50, 4 (March 2015), 399–411.
[39] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. 2015. *Accept: A programmer-guided compiler framework for practical approximate computing*. Technical Report UW-CSE-15-01. University of Washington.
[40] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 164–174.
[41] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. 2013. Approximate Storage in Solid-state Memories. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 25–36.
[42] M. Seo, C. Hong, S.-Y. Lee, H. K. Choi, N. Kim, Y. Chung, V. Umansky, and D. Mahalu. 2014. Multi-Valued Logic Gates based on Ballistic Transport in Quantum Point Contacts. *Scientific Reports* 4 (Jan. 2014).
[43] D. Shin and S. K. Gupta. 2010. Approximate Logic Synthesis for Error Tolerant Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 957–960.
[44] M. Själander, G. Borgström, M. V. Klymenko, F. Remacle, and S. Kaxiras. 2016. Techniques for Modulating Error Resilience in Emerging Multi-Value Technologies. In *Proceedings of the ACM International Conference on Computing Frontiers*. 55–63.
[45] M. Själander, N. S. Nilsson, and S. Kaxiras. 2014. A Tunable Cache for Approximate Computing. In *Proceedings of the IEEE International Symposium on Nanoscale Architecture*. 88–89.
[46] I. C. Society. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.
[47] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 1–14.
[48] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2013. Quality Programmable Vector Processors for Approximate Computing. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 1–12.
[49] S. Venkataramani, K. Roy, and A. Raghunathan. 2013. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1367–1372.
[50] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. 2011. MACACO: Modeling and analysis of circuits for approximate computing. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 667–673.
[51] H.-S. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (Dec. 2010), 2201–2227.
[52] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. 2012. Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. 441–454.

Received May 2018; revised May 2019; accepted XXX 20XX