Future Generation Computer Systems 113 (2020) 556–570

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

AxRAM: A lightweight implicit interface for approximate data access

João Fabrício Filho [a,b,*], Isaías B. Felzmann [a], Rodolfo Azevedo [a], Lucas F. Wanner [a]

[a] Institute of Computing, University of Campinas, Brazil
[b] Federal University of Technology, Paraná Campus Campo Mourão, Brazil

ARTICLE INFO

Article history:
Received 14 January 2020
Received in revised form 2 June 2020
Accepted 13 July 2020
Available online 17 July 2020

Keywords:
Approximate computing
Error tolerance
Memory interface
Approximate data

ABSTRACT

Approximate memories expose data elements to errors in order to improve energy efficiency. For a large fraction of data, these errors are inconsequential or lead only to small losses in application output quality. Nevertheless, for some critical data, errors may lead to execution flow crashes, resulting in non-produced outputs and wasted computational and energy resources. Thus, these techniques require some level of control over approximations to generate acceptable outputs and, consequently, to maximize the energy benefits. Many proposed interfaces for approximate memories rely on burdensome instrumentation of the program or on user annotations to protect critical data. We present AxRAM, a lightweight interface for approximate data that avoids crashes without user annotations. AxRAM relies on a memory with configurable reliability levels and protects from errors critical data regions commonly found on a number of applications. Furthermore, our interface implements a resilient addressing scheme that reduces invalid data accesses that lead to execution crashes. In an embedded computing scenario with a dual-VDD SRAM, our implementation of AxRAM reduces 51% of the execution crashes compared to an unprotected approximate memory, resulting in energy savings for 9 out of 12 profiled applications.

1. Introduction

Susceptibility to faults on modern hardware has increased with the technological scaling challenges in the dark silicon era [1]. Circuits are less reliable due to a higher exposition to transient faults, and mechanisms to suppress and correct these faults are increasingly costly with a negative impact on performance and energy efficiency [2–4]. At the same time, applications of data mining, classification, and synthesis have emerged as a significant portion of global computational resources and energy consumption, from mobile devices to large scale data centers [5]. For many of these applications that rely on massive volumes of data, exactness is not required or even possible. This creates opportunities for relaxing computational accuracy [6] across the system stack, from applications to circuits [7] in order to achieve better performance or energy savings [8].

Data approximation techniques explore storage in memory components at unreliable levels [9–12]. These levels lead to energy savings at the cost of possible errors in application data that may reflect in output inaccuracy [13–16]. Therefore, data elements as variables, constants, inputs, and references are exposed to errors, and applications executed in these environments should tolerate some inaccuracy in their results. Nevertheless, all applications have a limit of tolerated inaccuracy. As the number of data errors increases, the output degradation also grows and may trespass this tolerable limit, generating a useless result. Furthermore, these errors may affect some data differently from others in the same application. Function pointers are examples of critical application data — while an error in a pixel of a bitmap degrades a tiny portion of an image, an error in a function pointer may cause the loss of the control flow and, consequently, an execution crash without a result [16,17]. Without an output, computational efforts and energy resources are wasted, resulting in decreased benefits. Practical use of data approximation, therefore, requires interfaces that act between the application and the approximation technique, controlling what data can or cannot be exposed to errors, or triggering recovery strategies when critical errors occur.

Many data approximation architectures [10,11,18,19] propose different levels of reliability at memory partitions to store critical data in regions that are free from errors. However, the identification of critical data depends on several factors, such as the approximation technique, program inputs, data structures, and application context [20–23]. Several works [13,17,24–26] control approximations by relying on annotations or commands from the programmer to decide where to place data. While effective in protecting against crashes, these techniques bring additional

* Corresponding author at: Federal University of Technology, Paraná Campus Campo Mourão, Brazil.

E-mail addresses: joaof@utfpr.edu.br (J. Fabrício Filho), isaias.felzmann@ic.unicamp.br (I.B. Felzmann), rodolfo@ic.unicamp.br (R. Azevedo), lucas@ic.unicamp.br (L.F. Wanner).

https://doi.org/10.1016/j.future.2020.07.029
0167-739X/© 2020 Elsevier B.V. All rights reserved.

complexity, since programmers need to worry about the approximation control and must have expert knowledge about the application data. Moreover, an approximation-specific layer of code must be supported over the lifetime of the application, jeopardizing maintainability and portability. Approximation control interfaces that implement runtime systems, on the other hand, can monitor and recover from computations that cause execution crashes or quality lower than required [27–30]. Nevertheless, this type of system usually adds an execution overhead that decreases the approximation benefits on energy or performance by checkpoint and rollback mechanisms.

We present AxRAM, a high-level interface for approximate data access that improves execution resilience by allowing coarse-grained control of the approximate state of a data region. We found that invalid memory references are the main cause of crashes for many applications running on systems with approximate memories. To protect memory references without annotations in these data, we identify operations with invalid addresses and protect critical memory regions that store pointers. Thus, AxRAM corrects memory access violations by only accessing addresses within memory bounds, avoiding interruptions on the execution flow. To protect other references and critical data, AxRAM divides the data array in fixed-size memory banks, where each bank can choose between an approximate level and an accurate state. To isolate a memory region implicitly, we identify the system stack as an area that stores control pointers and other critical application data. Thus, our implementation isolates this memory location without any user annotation to identify critical data.

We propose an architecture model that allows the implementation of the AxRAM interface. This model is based on voltage overscaling approximation on an SRAM that has two global levels of the supply voltage. The first, higher, voltage level is the nominal value for the memory cell, which guarantees the execution of memory operations at the minimal designed error rate. An adjustable voltage regulator provides the second global voltage level, at which lower voltages lead to energy savings with higher error rates in the memory operations. Despite our interface is built to work with a voltage-overscaled SRAM, it is suitable to other approximate memories that exhibit related error models, such as DRAMs with timing or energy changes.

We evaluated the design by simulating the execution of 12 selected applications from various computing domains. For each application, we defined as error-free the minimum amount of banks to store the application stack, keeping locally allocated data safe and avoiding execution crashes on subroutine return. We detail our evaluation with an analysis of execution crashes, output quality, average energy cost, and quality-energy efficiency metrics. Our contributions, built upon our previous work in [31], include:

- An addressing scheme for data stored in approximate memories that avoids execution crashes;
- Implicit protection of a memory region that stores critical data;
- A memory architecture design that allows the coexistence of accurate and approximate memories of variable sizes in the same system;
- An exploration of applications and an analysis of how they behave under approximate environments.

Our experimental evaluation compares AxRAM with the use of a voltage-overscaled approximate memory, employing no data protection. Our results show that AxRAM eliminates data crashes, reducing 51% of total execution crashes. When comparing AxRAM with an approximate memory without any data protection, AxRAM offers energy savings of 9% at a 95% average quality threshold.

2. Background and related work

The Approximate Computing paradigm focuses on the exploration of error tolerance in applications to achieve performance or energy benefits [32]. From algorithm to circuit layers, several proposals [33–36] exploit error tolerance with an impact on execution flow and application data.

Approximation techniques provide opportunities to improve area, power, performance, and energy efficiency transferring accuracy constraints across the system layers [37]. Nevertheless, these benefits are limited to the acceptable inaccuracy of the application output.

2.1. Approximate data

Data approximation may lead to lower energy consumption or higher performance using techniques that approximate memory components [9]. Among several implementations of memory structures, most are highly sensitive to circuit variability since they typically use smaller components [15].

The diversity of components allows proposals of many memory approximation techniques, from the adjustment of operating parameters [16,33,38] to modification of hardware components [39,40]. Modifications of hardware components include replacing blocks with similar data [40]. Parameter adjustments depend on the type of memory and its features, like DRAM refresh rates [33], PCM density [16], and SRAM supply voltage [38,41].

Voltage overscaling is an approximation technique that modifies the supply voltage to values under nominal specification [42]. A memory circuit in this condition exposes its data to noise that may cause dynamic and static errors. This instability follows the voltage scaling on SRAMs regularly, which leads to a relation between error rate and supply voltage [38].

AxRAM experimental implementation uses dynamic voltage scaling on SRAM components to achieve energy savings at the cost of data error exposition. Our estimation of energy savings follows the model proposed by Wang and Calhoun [38], that relates the SRAM supply voltage to an error probability.

2.2. Data protection interfaces

Interfaces that control approximations at data level usually separate application data into error-resilient and accurate. This separation is necessary to avoid errors in application data that are critical and do not tolerate errors. Errors on these data may nullify execution results, decreasing the average quality of outputs and increasing the energy consumed by recovery mechanisms. Thus, errors in critical data also impact the benefits provided by the approximation technique.

The protection of critical data can be performed through annotations on variables at high-level programming language. EnerJ [13] features type qualifiers that split data into two levels of data quality: approximate and precise. This demands architectural support for these two levels of operation. Energy Types [24] allows more levels of operation through energy specifications. Despite the higher control of the programmer, this proposal also demands hardware support for this regulation. DECAF [20] allows error tolerance degrees and extends quality constraints to non-annotated data. It proposes a compilation system that infers data affected by annotated data. This allows the programmer to annotate only the most crucial data. DECAF reaches this implication through static and dynamic analysis of the application.

When the approximation mechanism is applied directly to the hardware, the application requires some interface to control (or be controlled by) the approximations. Previous work usually implements such control in the architecture level, extending the

ISA to identify and protect critical regions and define the execution flow. Truffle [11] is a micro-architecture implementing dual-voltage operation that supports ISA extensions for data protection, where critical data operations are executed in a precise way. Quora [19] is an extension to ISA that allows choosing quality constraints explicitly on hardware instructions through programmable vector processors. STAxCache [26] combines circuit and architectural techniques to control the impact of errors produced by a cache memory implementation controlled by user instructions. This ISA extension allows the user to specify quality requirements on data arrays of the cache memory.

Our architectural model is based on a two reliability level memory design, where it is possible to store data less resilient to errors in the more reliable level of operation. AxRAM protects critical application data without requiring the user to identify them by protecting the region of the system stack, which commonly contains critical data in many types of applications.

2.3. Runtime system interfaces

Runtime systems control and recover from errors by dynamically managing the execution flow. These interfaces usually require instrumentation or tuning to add checkpoints and recovery code to avoid execution crashes. Error management at runtime aims to avoid unexpected execution behaviors by dynamically capturing the execution flow. Relax [25] is an architectural framework that offers runtime control to software recovery of application behavior. It includes an ISA extension that allows the compiler to guarantee the state of the program through retrying or discarding computations. Ringenburg et al. [30] propose a dynamic quality monitor with offline debugging instrumentation that tracks data flow of approximate operations. The quality monitor uses correlations identified by the offline tool between individual operations and output quality.

Recalibration of online monitoring sometimes is necessary to update quality measures or error control. SAGE [27] is a runtime system with compiler support to generate several kernels with distinct levels of reliability. Periodically, the runtime system is calibrated to dynamically choose the appropriate kernels. Rumba [28] recovers from large errors detected by simple prediction models and measured through online parameter tuning. ApproxQA [29] considers an approximation mode tuning through learning algorithms and allows recovery computations based on a statistic accuracy threshold. Crash Skipping [43] avoids overheads of recovering computations by skipping instructions that lead to crashes. If the current instruction causes the interruption of the execution flow, it is replaced by a `nop`, and the control flow continues sequentially.

The addressing scheme of AxRAM is a runtime system that does not need checkpoints, recalibration, or tuning. This makes the interface a lightweight system that does not add significant overheads in the execution environment or user annotations. Crash Skipping [43] is an interface that fits in these requirements, but without any attempt to recover incorrect data. This interface also avoids crashes caused by execution flow deviations, redirecting to the last executed function or instruction. AxRAM avoids redirecting the control flow as a feature since operations that cause these crashes would deviate the control flow to an unrecoverable memory region. Furthermore, AxRAM treats incorrect pointers that cause crashes and protects regions of critical data. Thus, our interface avoids execution crashes caused by out-of-bounds memory addresses and corrects these data to values within the bounds, unlike Crash Skipping, which just avoids crashes.

3. The AxRAM memory interface

AxRAM is an interface to improve execution resilience, maximizing the benefits provided by approximate memories. Our proposal improves the average output quality to increase energy efficiency by avoiding execution crashes in approximate data environments. This approach considers that, in a production scenario, an application typically runs multiple times with different inputs. Every single execution instance is subject to some errors from the approximate environment (in this case, the memory), which may lead to quality degradation and, eventually, an execution crash. By avoiding crashes, we allow many of the previously unsuccessful execution instances to last longer and produce some output. Thus, the average quality amongst a whole batch of executions is increased not by better quality for every single instance, but mainly by producing more results. This reduces the amount of energy spent on unsuccessful computations, increasing efficiency, and potentially allows the whole application to be subject to higher error levels, as discussed in our experimental results.

We propose two modifications into memory design to avoid execution crashes. Our improvement focuses on execution resilience by (1) treating accesses out of allowed memory boundaries and (2) protecting critical data regions commonly found on many applications. The remainder of this section classifies execution crashes and discusses the fundamentals of these two approaches.

3.1. Types of crashes

Execution crashes are premature terminations of the execution flow that lead to no output production. Without an output, the quality cannot be computed and is perceived as zero, which reduces the average quality of executions. These terminations usually are caused by errors in critical application data. We classify execution crashes into three types:

- **Data crashes**: when an attempt to fetch data from an invalid memory address causes an access violation.
- **Flow crashes**: when the control flow tries to jump to an incorrect region of memory.
- **Timeouts**: when there is no valid result produced after some reasonable, application-specific, amount of time.

A `load` or `store` operation with an out-of-bounds pointer causes a data crash. An attempt to jump to an invalid control address causes a flow crash. Timeouts happen on applications that rely on data convergence, when errors accumulate and prevent the execution to meet the stop criteria. Besides, the use of data structures based on memory references may cause these crashes: an error may produce a wrong pointer, causing infinite iterations over random or irrelevant memory locations.

3.2. Treatment of incorrect pointers

Memory boundaries are defined by the size of the memory in embedded systems and by the application bounds in virtual memory. A significant number of crashes are caused by memory operations on addresses that are out of application boundaries. These addresses were data pointers, stored in memory, that had one or more bits flipped due to an error. An attempt to operate with an invalid address makes the system throw an access violation signal. This signal stops the control flow and discards the remaining computation, causing an execution crash. This leads to no output being produced and decreases the average quality of results in the approximate environment.

To correct invalid addresses, we need to identify data pointers stored in memory. Nonetheless, pointers are indistinguishable

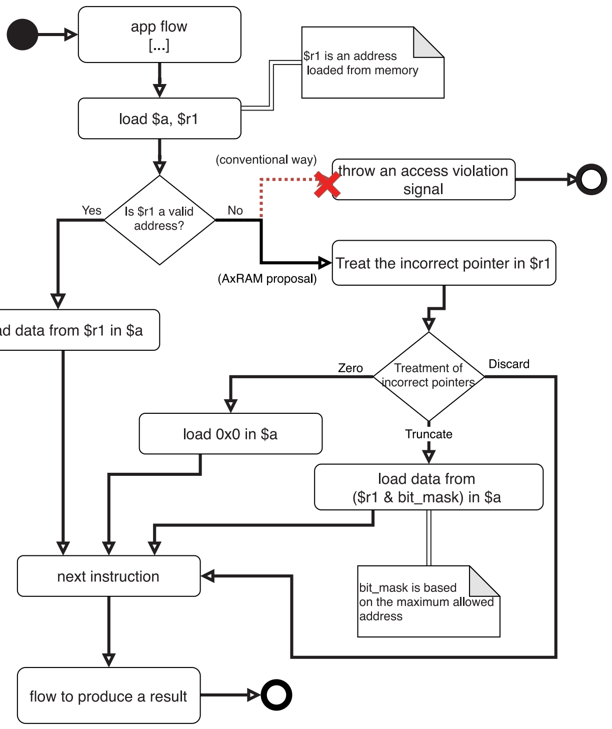

app flow
[...]
$r1 is an address loaded from memory
load $a, $r1
(conventional way)
throw an access violation signal
Yes
Is $r1 a valid address?
No
Treat the incorrect pointer in $r1
(AxRAM proposal)
load data from $r1 in $a
Zero
Treatment of incorrect pointers
Discard
load 0x0 in $a
Truncate
load data from ($r1 & bit_mask) in $a
next instruction
bit_mask is based on the maximum allowed address
flow to produce a result

Fig. 1. Treatment of incorrect pointers working in a `load` instruction.

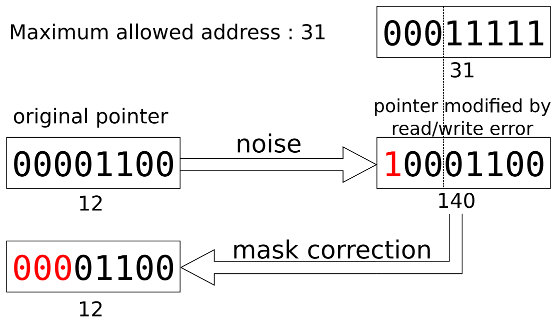


Fig. 2. Example of how the memory boundaries protection works.

from any other data at the memory level, and identifying them requires additional information from the application level, adding significant overhead. Hence, we propose to detect data pointers through instructions that manipulate pointers. When `load` or `store` instructions are executed, these instructions receive a data pointer as a parameter.

Our proposal is an addressing scheme to treat invalid pointers that are out of allowed memory boundaries. AxRAM identifies data pointers on memory operations and verifies whether these addresses are within memory bounds or not. Instead of throwing an access violation signal, we proceed with the computation after treating the incorrect pointer. We evaluate three forms of treatment for incorrect pointers during the execution: (1) discarding the current instruction, (2) zeroing the destination register, and (3) truncating the address indicated by the pointer within bounds.

Fig. 1 shows an example of how is the workflow of the treatment of incorrect pointers with a `load` instruction. Depending on the implementation of this treatment, different values are loaded into the destination register, but, in all cases, the execution flow continues to the next instruction. This is different in the conventional way, where the execution flow stops and throws an access violation signal. However, in the case of a `branch` instruction, the incorrect pointer refers to the next instruction in the execution flow and, therefore, the remaining instructions may be lost after a flow crash.

Discarding the current instruction leaves unchanged the value in the destination register, and the remaining computation proceeds without any change in context. This treatment can be advantageous in the case of a loop that uses the destination register to load temporary values, for example, because the computation proceeds with a value from the previous iteration. If these values are pixels of an image, the value from a previous iteration can be similar to the current one.

Loading zero in the destination register can be advantageous in the case of some data structures like linked lists. These structures depend on pointers that indicate the next element. A pointer reading zero is conveniently used to indicate the end of the list. When finding an incorrect address while iterating over such a structure, there is no way of finding where the next position is stored, thus zeroing the value would indicate the end of the list and allow computation to proceed towards some, not necessarily correct, output.

Finally, truncating the pointer value is a more generic approach. The truncation applies a mask to the value based on the characteristics of the memory space. All bits that would represent an invalid memory location are zeroed, forcing the address to be valid. This truncation is an attempt to correct the address to the original value, considering the common case represented in Fig. 2, in which the memory is smaller than the addressing capability of the data word, where invalid addresses would be triggered by an incorrect reading of higher magnitude bits of the address. This treatment can be advantageous in more general contexts since it tries to recover the original value of the pointer. Nevertheless, there is no guarantee that an incorrect pointer will return to its original value after the mask is applied because an error may occur on less significant bits or more than one error may change the pointer. In the case of an incorrect (but valid) pointer, the computational work proceeds with the wrong data fetched from this address.

The choice for which treatment to perform depends on the implementation of the interface. To simplify the usage of this feature, AxRAM loads the configuration at boot time and performs the same treatment for all applications. Thus, the treatment of incorrect pointers requires no user intervention since the execution environment triggers it automatically. This addressing scheme can be implemented on the architecture, OS, or as a runtime system that encapsulates memory accesses. AxRAM implements an error recovery mechanism that does not need checkpoints or program instrumentation. Furthermore, the implementation of AxRAM as a runtime system represents a lightweight avoidance of and recovery from crashes without programmer intervention or program modifications. The energy cost of this implementation is lower than runtime systems that recover from errors by checkpointing, monitoring the execution, and re-executing entire functions or computational tasks. Moreover, the hardware implementation of this treatment is as simple as an AND gate in the memory input to modify pointer values, which represents a negligible overhead in performance or energy.

3.3. Critical data protection

Incorrect data pointers cause a significant part of execution crashes. Nevertheless, these pointers are not the only critical application data. Flow pointers or control indexes, e.g. return addresses of functions, file headers, and loop control indexes, are critical values that are not treated by the addressing scheme.

Several works [13,16,20,24,44] propose the error isolation of critical data by extending the programming language to include annotations to classify how critical each data portion is. These annotations require, from the programmer, expert knowledge of the approximated environment, of the control mechanism, and a full understanding of the application data, reducing the portability of the solution. Thus, the automatic identification of critical data is essential to improve execution resilience without programmer intervention.

We identify the system stack as a contiguous region, usually small, that contains some critical data in many types of applications. Compilers use the system stack to store temporary execution values, such as shorter-lived automatic variables and loop control indexes. Errors on these indexes may cause a loop to execute indefinitely, which leads to a timeout crash. Furthermore, return addresses of functions are commonly stored in the stack region. These pointers indicate where the execution flow must return after the end of a called function. An error on these data makes the execution flow try to jump to an incorrect address causing a flow crash immediately. Furthermore, the system stack boundaries are easily traceable at the architecture level. These characteristics make the system stack a natural candidate region to be protected from errors without programmer intervention.

4. Implementation

A memory interface acts between an approximation technique and the application. The approximation technique depends on an environment implementing an architectural model that allows the approximation. In this section, we discuss the implementation of our interface with approximation techniques and other issues that could be faced on the usage of AxRAM.

4.1. Architectural model

AxRAM offers two main features to protect and recover an application from crashes in an approximate environment. The treatment of incorrect pointers operates directly at the memory addressing scheme, independently from the architectural model. Nevertheless, critical data isolation requires some architecture support to isolate some parts of the memory from errors.

AxRAM architecture model for data isolation defines two reliability levels in memory storage. One of these levels should be an operation considered as free from errors. Several proposed architectures [10,11,18,19] separate memory regions by reliability levels to isolate some data from errors. AxRAM is compatible with the architectural model of Truffle [11]. This model, however, requires the usage of ISA extensions, which demands changes in some level of the application. Therefore, we purpose an architectural model to avoid excessive application changes.

Our architectural model also uses voltage scaling as an approximation source in an embedded SRAM main memory. Despite that, AxRAM is suitable for other memory approximations that exhibit non-deterministic data errors and allow the division of memory into approximate and non-approximate regions. In our architectural model, illustrated in Fig. 3, SRAM is divided into banks, where each memory bank defines a region supplied by a common voltage. One single configurable voltage regulator supplies a voltage level below the nominal specification of the memory banks, that is, a voltage level at which the stored data is more susceptible to errors [38]. A switch in the power line of each bank specifies whether the bank should use the nominal, higher, error-free V_{in} voltage or the lower error-prone voltage supplied from the regulator. The additional hardware has negligible impact in performance and area overhead by power-gating transistors and small inverters that represent less than 1% of area overhead [45].

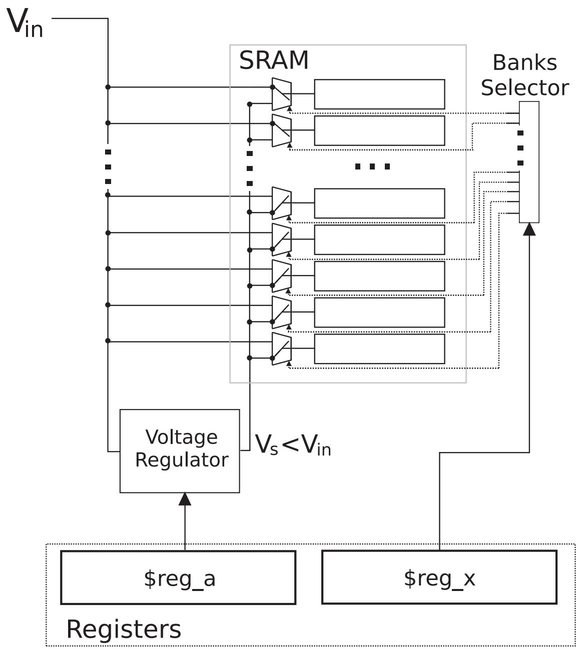


Fig. 3. Memory architecture of AxRAM.

To control the SRAM approximation, the model includes two memory-mapped registers. Register $reg_a defines the voltage level supplied by the voltage regulator and register $reg_x controls the gate switching to define which banks are on the approximate state.

These memory-mapped registers work as knobs to control approximations. In the case of an embedded systems environment, without the supervision of an Operating System (OS), these knobs can be configured before the execution of the application. Thus, the main work to port an application to this environment is to find the tolerable error rate for the application. This error rate represents the limit of imprecision tolerated.

4.2. OS support

The architectural model from 4.1 supports the execution without application changes in an embedded systems environment with AxRAM as a runtime system. If an OS supervises the application, the control knobs should be configured by the OS since other applications could use the same memory. The OS is the runtime system that implements the interface, in this case.

To make the configuration of the approximate memory simpler, our architectural model supports only two reliability levels concurrently — one considered as precise and another that exposes data to a configurable error rate. Thus, all data stored in approximate regions are exposed to the same error rate at a given point in time. This error rate is the same for all applications that run in the same environment and whose data is exposed to errors. Therefore, a syscall with a probability as parameter should be available to configure the SRAM error rate when necessary.

The reliable regions of the approximate memory contain the program stack, which belongs to an application in an environment with an OS. The application stack is previously allocated by the OS. Thus, to implicitly protect this region, the OS has to specify in the $reg_x register the memory area reserved to store the stack. The OS has control over all memory pages allocated to

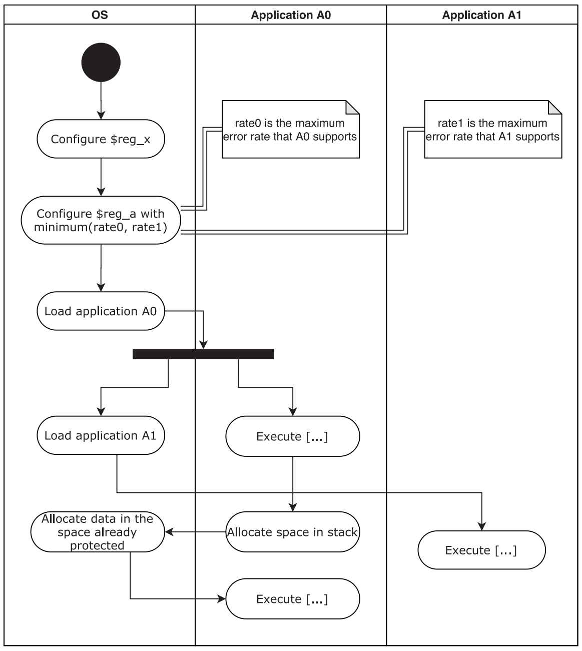

OS
Application A0
Application A1
Configure $reg_x
rate0 is the maximum error rate that A0 supports
rate1 is the maximum error rate that A1 supports
Configure $reg_a with minimum(rate0, rate1)
Load application A0
Load application A1
Execute [...]
Allocate data in the space already protected
Allocate space in stack
Execute [...]
Execute [...]

Fig. 4. An example of the OS workflow running applications A0 and A1.

the applications, thus the stack protection works without changes in the application.

Fig. 4 shows an example of the workflow of an OS running two applications, where it prepares the execution of each application pre-allocating error-free regions by configuring $reg_x. If more space in the application stack is needed, it is allocated in these regions. The error rate of the approximate regions, in $reg_a, is configured as the minimum supported value between all applications.

The second main feature of AxRAM is an addressing scheme to treat out-of-bounds memory references. If the application is supervised by an OS implementing virtual memory, this scheme considers only the virtual address space. The addressing scheme is activated when the system is going to raise an access violation signal. Instead of treating it as a segmentation fault, the OS applies the treatment of incorrect pointers and continues with the remaining computation. The proposed treatments by discarding the current instruction or considering as zero the current data are implemented independently from OS addressing. Nevertheless, the truncation of the incorrect pointer should consider the virtual memory space of the application. Since all memory references within an application are virtual memory addresses and the correction of the pointer forces it to within the specific application virtual memory space, the approach does not affect memory isolation, nor does it influences any other memory management activities.

4.3. Concurrency control

AxRAM critical data protection works on a configured error isolation of memory banks of $reg_x and an approximation level control of $reg_a. The isolation of memory banks from errors allows control over which parts of data are not exposed to errors. AxRAM proposes to isolate the application stack, making this control application-specific. Thus, the execution of more than one process does not affect this protection. However, to multi-threaded applications, several flows are running using the same error-free region. Thus, the control of this concurrency in shared memory regions is necessary but it is the same as without the approximate environment.

The error rate, in the proposed architectural model, is the same for the entire approximate memory, so multiple processes or threads running in parallel are exposed to the same error rate. Therefore, if applications with different error tolerance are executed in the same environment, the approximation level should be adjusted to the minimum tolerated error rate of these applications. This decreases the energy benefits of the most error-tolerant application but allows the execution of the less tolerant.

5. Methodology

This work relies on the study of errors impact at the application level. This study needs to evaluate different techniques and error models. A fast alternative to make this evaluation is the modeling of data errors at higher-level abstractions in a simulation environment. A simulation environment allows implementations of different approximation techniques on several technologies.

5.1. Implementation and setup

The experimental evaluation of our proposal is in an embedded systems environment where one single-threaded application runs in bare metal in the CPU without the supervision of an OS. In this environment, the application has access to the entire memory array through an SRAM main memory with the architectural model described in 4.1. This implementation is a model with no influence of other applications or a middleware on the effect of errors and on energy savings.

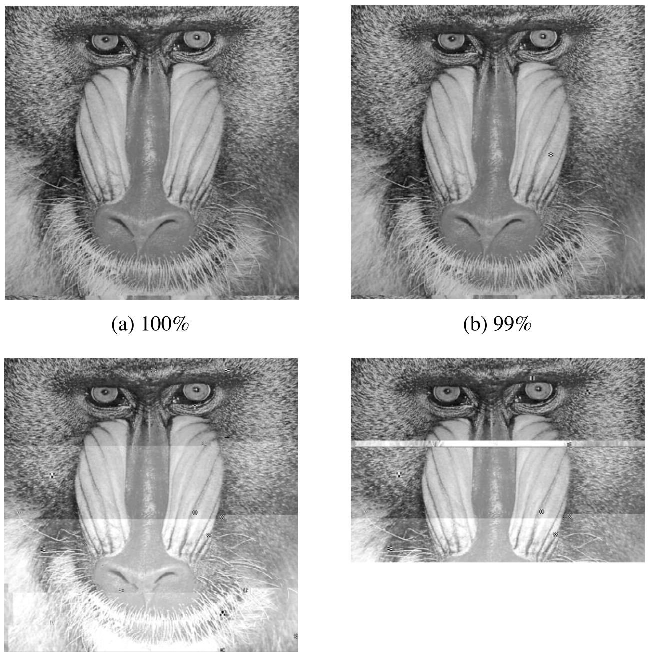
In our modeling approach, an approximate state changes the supply voltage of the data memory and exposes data to dynamic errors according to a uniform probability. This model is implemented in an ADeLe-generated [46] CPU model for the ArchC architectural simulator [47] using a MIPS32 architecture. The simulator replaces all read and write operations on the data memory with a software model that is susceptible to error, by performing a single bit flip in a random position of the data word. This bit flip represents an error according to a uniform probability specified in $reg_a.

We compare AxRAM with an approximate memory in a scenario of a voltage-overscaled SRAM that implements the same approximate states, architecture, and error model [38]. This environment implements neither AxRAM addressing scheme nor critical data protection. The implementation of the same approximate states makes this a fair comparison because it allows the applications the same data error probability on both techniques, with and without our proposed features. We refer to this environment as “approximate memory” in our results section.

5.2. Quality metrics

Quality metrics try to quantify how different is an approximate output compared to the accurate output. Thus, the quality metric is resulting from a comparison between the outputs of accurate and approximate executions of the same code with the same input. The metrics that we use in this work are:

- Mean Relative Error (MRE);
- Normalized Number of Equal Elements (NEE): The fraction of elements (single datum, lines, or words) in both output data.
- Normalized Number of Elements Out of Margin (NEOM): The fraction of elements out of an acceptable margin of error.
- Structural Similarity (SSI): The difference between two images [48].


(a) 100% (b) 99%

(c) 76% (d) 49%

Fig. 5. Examples of SSI quality measure of images.

All of these metrics return a percentage indicating how similar the application output is compared to a non-approximate execution, where 100% means they are identical. While different metrics are not numerically comparable, the normalized range enables us to define a unique threshold for all applications for a better understanding of the results. Furthermore, different quality metrics can be used to evaluate the same kind of data, such as MRE and SSI measuring the difference between images. Some metrics, however, are restricted to specific data types, as SSI to images. In this work, we chose one quality metric per application to simplify the evaluation of execution outputs.

Fig. 5 exemplifies how the occurrence of memory errors affects the output of the application jpeg and how the quality metric SSI captures the quality depreciation. The first image shows an error-free output, in which the computation result is identical to the accurately-computed baseline. Images (b)–(d) show two types of consequences of errors, depending on the code region they affect: incorrectly storing parts of the Huffman code into memory cause errors isolated to single 8×8 pixels blocks, causing a lower impact on similarity; and incorrectly retrieving the Huffman coefficients from the last computed block, where the coefficients are cascaded, causes the image to exhibit stripes of different brightness, leading to a larger impact on similarity. Since errors are injected randomly, they can happen at any point in the execution, and the final impact on quality is not directly a function of the error rate. For example, both images (c) and (d) were subjected to the same error rate of 1.27E−5, but in the second case a high brightness causes the image to appear white at the bottom, thus the difference from the original image is more evident and the quality metric is lower.

5.3. Applications

Table 1 shows the applications we use on the evaluation. These applications represent a wide range of the usages of computational systems, such as linked lists, function pointers, floating points, compressing, and arithmetic operations. Furthermore, some applications represent different implementations of a solution to the same problem, like dijkstra and floyd–warshall (the shortest path problem), or manipulate the same kind of data inversely, like bzip2 and bunzip2 (compressing/decompressing). We classify applications into three types: signal processing, CPU-bound, and memory-bound. Signal processing applications are commonly applied to the context of approximate computing and include image processing applications. CPU-bound applications are kernels that stress the CPU in the majority of its processing with few memory accesses. Memory-bound applications spend major time of the execution with accesses to memory into the application main computational stage, to which we refer as kernel.

Table 1
Applications of our experiments.

Application	Type	Quality metric
2mm [49]	Memory-bound	MRE
nbody [50] spectralnorm [50]	CPU-bound	MRE
reg_detect [49]	Signal processing	MRE
bunzip2 [51] bzip2 [51] dijkstra [52] floyd–warshall [49] qsort [52]	Memory-bound	NEE
fft [52]	Signal processing	NEOM
jpeg [53]	Signal processing	SSI
mandelbrot [50]	CPU-bound	SSI

Being voltage-overscaling a non-deterministic approximation technique [7], we need to perform several executions to evaluate the impact of errors in each application. Thus, we execute 100 times each application at each error rate. The error rate determines the approximation level of the technique. We evaluate the applications at 40 error rates in logarithmic intervals from 1E-9 to 1E-4. The 1E-9 represents an error rate where most of the evaluated applications do not have execution crashes, and, in the 1E-4 error rate, most of the evaluated applications obtain results in which quality equals zero. For simulation purposes, the approximation technique was applied only to the code in the kernel of each application, in order to avoid errors to happen during I/O phases that emulate some peripheral behavior.

5.4. Quality control and energy

The usage of approximate memory implies errors in some application data. These errors can result in some quality loss in the output of each execution. Since the minimum acceptable quality depends on the context of the application, the tolerable limit of error also depends on this context. Nevertheless, there is no way to measure the output quality without the accurate result, obtained through a non-approximate execution. To calculate the energy savings, we define a threshold in the quality metric that each execution output must obey. Thus, the energy cost is based on several quality thresholds of execution outputs. To avoid application stalling, we set a timeout as the double of the accurate execution time for each application.

The energy savings are based on a relative value to the nominal voltage of the SRAM. The data to infer the relative voltage is extracted from Wang and Calhoun [38], where the authors present error rates for a voltage range of SRAM cells calculated through the static noise margin of these cells. The data used in this work is from a 45 nm 6T SRAM, balanced cells where the error rate is independent of the stored data. We implement the errors in memory read and write operations considering the rate of the highest error probability at each voltage.

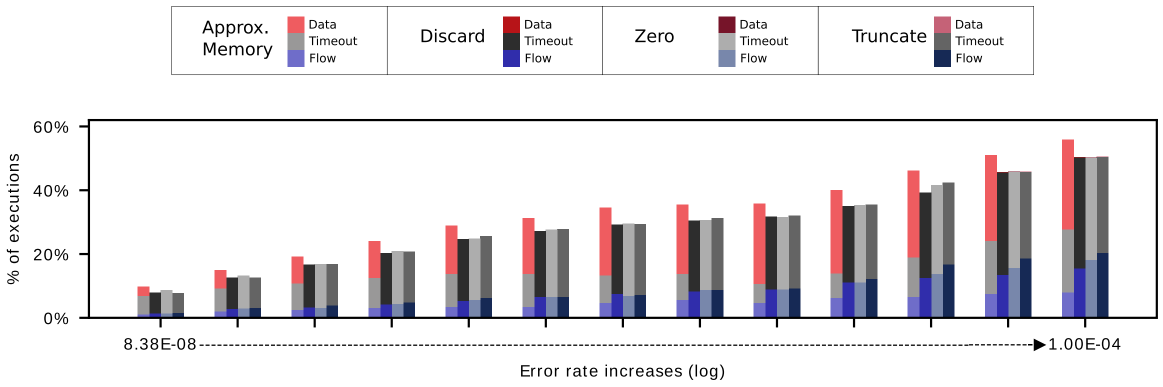


Fig. 6. Executions crashes implementing techniques to treat incorrect pointers.

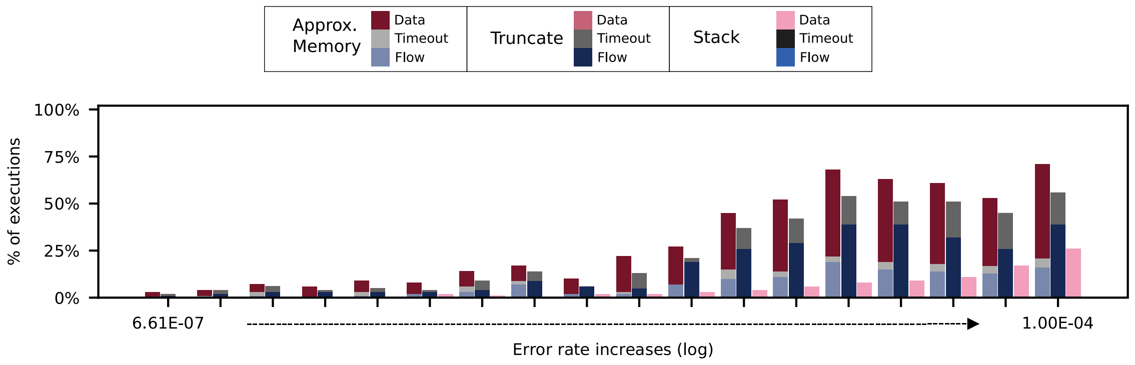


Fig. 7. Jpeg: Executions crashes implementing AxRAM protection techniques — No crash happens combining both *Truncate* and *Stack*.

6. Evaluation results

We present in our experimental evaluation a comparison between the three forms of treatment of incorrect pointers. Further, we consider the treatment by truncation as the implementation of AxRAM in our chosen environment and illustrate in a case study the analysis of the impact of addressing and data protection on execution crashes. Lastly, we evaluate and discuss AxRAM considering four main metrics: number of execution crashes, output quality, energy savings, and quality-energy efficiency.

6.1. Treatment of incorrect pointers

The purpose of this treatment is to avoid data crashes caused by incorrect pointers. Fig. 6 shows execution crashes of all applications with the evaluated error rates to the three forms of treating incorrect pointers in isolation. *Truncate* refers to the treatment by truncating incorrect pointers into allowed memory boundaries, *zero* refers to writing the value zero in the destination register, and *discard* refers to the treatment by discarding the instruction with the incorrect pointer. The left-most bar shows the crashes for the use of an approximate memory without any protection or treatment.

As the error rate increases, the number of execution crashes also grows, however to a lesser extent when applying techniques for the treatment of pointers. The three addressing schemes have similar behavior considering the type and the number of execution crashes. Data crashes are almost zero among all applications, while flow crashes and timeouts increase together with the error rate. However, *truncate* exhibits fewer timeouts and more flow crashes than the other treatments among several error rates. Timeouts are energy costly executions that do not produce any result. To avoid such wasted resources, and given that the three techniques are similar in other aspects, we focus our evaluation on an implementation of AxRAM that uses *truncate*.

6.2. Impact of addressing and data protection

The addressing scheme and the stack protection of AxRAM intend to reduce execution crashes that the accesses to incorrect memory addresses cause. Fig. 7 shows the observed crashes for a case study application in three chosen scenarios. *Approximate memory* refers to a voltage-overscaled approximate memory without AxRAM protections. *Truncate* refers to the use of the addressing mask to treat incorrect pointers by truncation, in isolation. At last, *stack* refers to the AxRAM implementation of critical data isolation with stack protection only. We omit the results of AxRAM implementation with both techniques since this scenario eliminates all crashes in the studied application.

The trending scenario of crashes is that the higher error rates determine the higher number of crashes. Nevertheless, the errors are non-deterministic and may occur at any point in the execution. An error at a critical point may cause an execution crash. Therefore, some higher error rates may show a smaller number of crashes, but without effects on the trending line.

The addressing mask of *truncate* corrects only pointers that would fall into invalid memory locations because of a memory error when fetching the pointer. When a pointer read from memory contains some error but still falls within a valid memory region, this error is undetectable by the interface and the execution proceeds as if no error had happened. Thus, although valid, the address may point to a memory location that does not contain the expected value, causing some consequence according to how this value is used during execution:

- The value is used as part of a data region of the application (e.g., a pixel of an image for jpeg), the computation proceeds with the wrong value, and some quality degradation is perceived. This is the most common behavior perceived in our test applications, where data crashes are eliminated, and the execution proceeds with impact in quality;

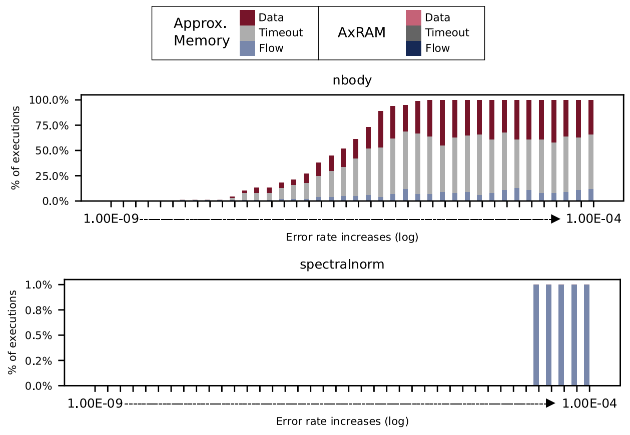


Fig. 8. Crashes of CPU-bound applications.

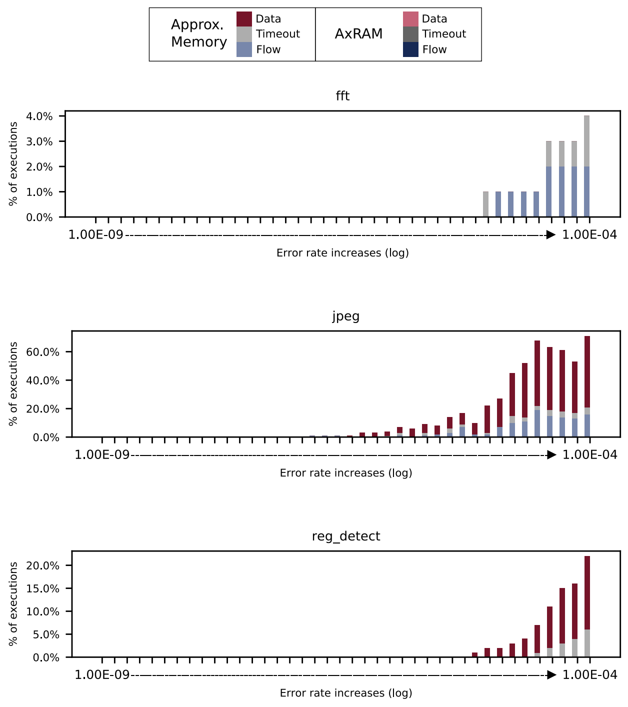


Fig. 9. Crashes of signal-processing applications.

- The value is used as a reference to a code region (e.g., a function pointer) and the application jumps to the incorrect address, breaking execution flow and/or causing a flow crash. Our experimentation scenario includes applications that make use of function pointers and exhibit flow crash behavior, such as qsort;
- The value is used as a reference to another data region (e.g., a data pointer in a linked list) and the application fetches and uses data incorrectly, causing quality degradation or, eventually, entering an infinite loop, a timeout crash. Our experimentation includes applications that make use of data structures that rely on data pointers and, thus, exhibit this timeout crash behavior, such as dijkstra.

The addressing scheme of *truncate* eliminates data crashes, while these crashes interrupt 50% of executions in *approximate memory* evaluations at a 1E-4 error rate. Nevertheless, flow and timeout crashes sum 21% of *approximate memory* executions and 56% of *truncate* executions. The causes of more crashes of these types are the result of the scenarios discussed above. Nevertheless, the number of successful executions on *truncate* is 44% in comparison to 39% on *approximate memory* at the maximum error rate evaluated.

The stack protection implementation achieves an aggressive elimination of crashes compared to both other techniques. *Stack* eliminates flow and timeout crashes at all error rates. On this technique, the interruptions due to data crash are 26% of executions at maximum error rate, while *truncate* eliminates data crashes at this error rate, but exhibits 17% and 39% of timeout and flow crashes, respectively.

The evaluation demonstrates that *stack* and *truncate* attack different types of crashes and in different ways. The combination of both techniques eliminates all crashes on the studied application. Therefore, it shows potential improvements in application resilience in the evaluated scenario.

6.3. Execution crashes

Execution crashes cause premature interruptions in the execution flow, without producing an output. Thus, an execution crash implies a zero quality output and increases the energy cost since a new execution is necessary to recover the lost data. The AxRAM protection cannot eliminate execution crashes for all application domains. Especially when applications heavily rely on memory references or convergence, an execution flow deviation may lead to an interruption by timeout. In the use of data approximation, the number of crashes tends to increase with the error probability. There are downside deviations at some points, because of the non-determinism of the errors. A higher number of executions may soften these deviations.

AxRAM shows no data crashes at any error probability in our evaluation. The addressing mask of our design avoids data crashes due to truncating out-of-bounds addresses instead of crashing the execution. To analyze the behavior of all applications, we separate the analysis by application type. Applications of the same type have some characteristics in common, but this does not determine their crash behavior.

CPU-bound applications exhibit fewer crashes than other types due to a non-intensive use of memory, the approximate component. Fig. 8 shows the execution crashes of these applications. In general, the use of *approximate memory* does not strongly affect CPU-bound applications since just nbody shows a significant number of execution crashes in this environment. Nbody uses data pointers to iterate over its vectors, which explains the high occurrence of data crashes. The use of AxRAM highly benefits nbody, once all executions produce results up to the 1E-4 error rate.

Fig. 9 shows execution crashes to **signal-processing applications**. These applications have many error-tolerant data but crashes affect their executions more than CPU-bound applications in *approximate memory*. AxRAM eliminates all execution crashes in the evaluated error rates to these applications.

Memory-bound applications are commonly more susceptible to crashes due to the intensive use of memory. Fig. 10 shows crashes for these applications. The non-determinism of crashes strongly acts on bunzip2 due to the error recovery mechanisms in its kernel. The application bzip2 has similar operations than bunzip2 but does not have the error recovery mechanisms and is more affected by execution crashes. Dijkstra and qsort also represent corner cases in the AxRAM technique. Dijkstra uses a list-like structure that strongly relies on pointers to store its information. If one of these is incorrectly read, the application

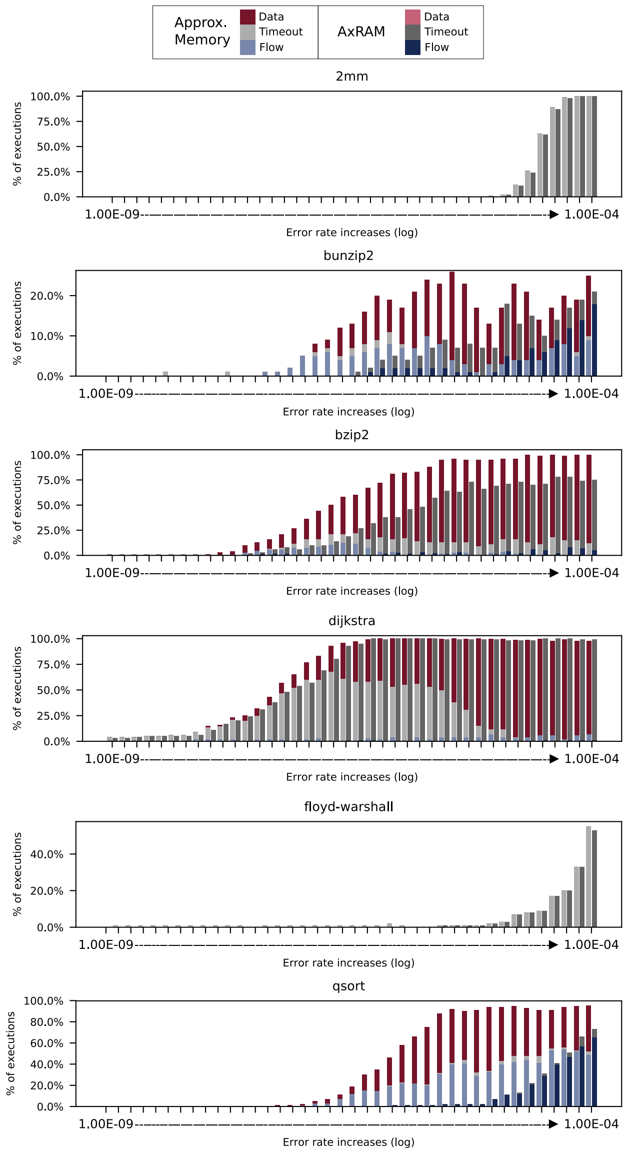


Fig. 10. Crashes of memory-bound applications.

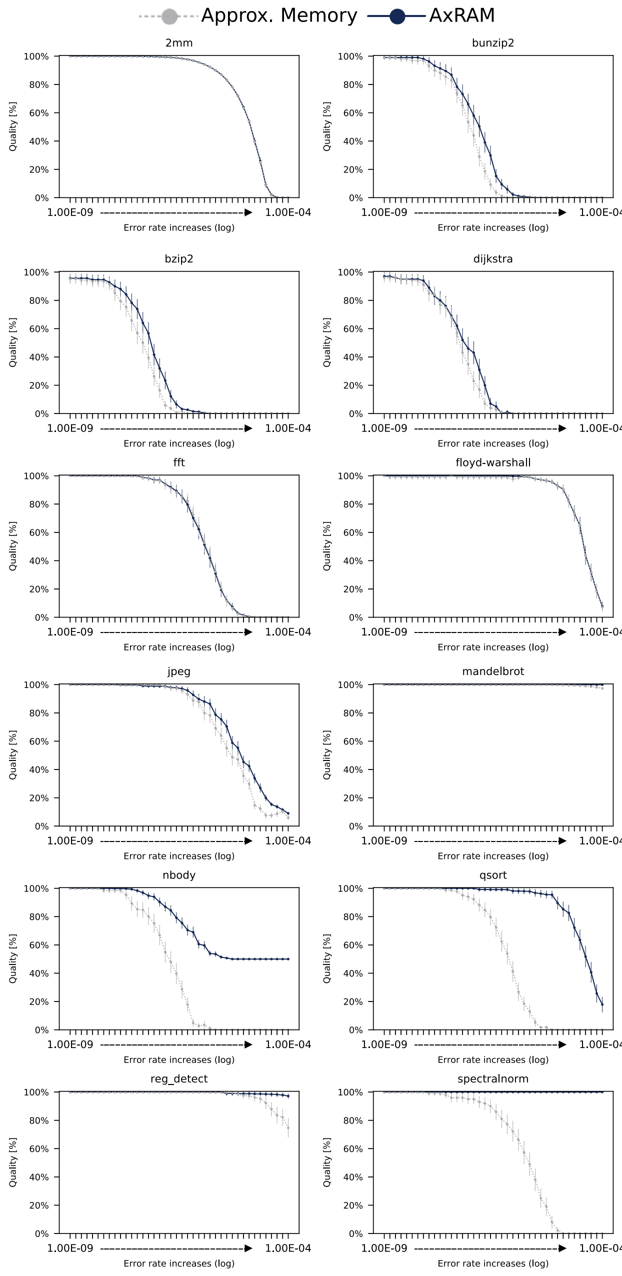


Fig. 11. Average quality of each application.

would loop over random data in memory, causing a timeout crash — thus although many data crashes are eliminated, timeouts take their place. Qsort uses function pointers to call the comparison routine within the sorting algorithm. Differently from data pointers, these are not recovered by the addressing mechanism and end up causing flow crashes. These are more noticeable in the AxRAM scenario because, after eliminating data crashes, the application ends up lasting longer and increasing the probability of an error to affect a function pointer. Memory-bound applications are in general strongly affected by execution crashes in *approximate memory*, and AxRAM protections show the potential to decrease and postpone significantly the number of crashes in three applications: bunzip2, bzip2, and qsort.

6.4. Quality

Execution crashes highly influence the average quality since each one represents a null-quality output. Nevertheless, errors in non-critical data influence the quality degradation as well and, therefore, also impact the average quality. When compared to *approximate memory*, AxRAM allows a higher error rate to achieve the same expected average quality for the application output, which potentially translates into higher energy savings. Fig. 11 shows the average output quality to all executions of evaluated applications. Since each error rate point in the X-axis is a logarithm interval, a small displacement to the right that AxRAM allows in the curve represents several degrees of energy-quality adjustment. The results show that 8 out of 12 applications exhibit significant quality improvements with AxRAM.

CPU-bound applications show different patterns of the quality line behavior. Mandelbrot has an almost null impact with the use of approximate memories. Spectralnorm executions show lower quality without a significant number of crashes. Nbody gets low quality early on an abrupt fall because of execution crashes.

AxRAM is capable of postponing the quality decrease of nbody and holds up the quality of spectralnorm.

Signal-processing applications usually tolerate more errors than other applications. The behavior of jpeg and reg_detect applications with AxRAM protections is the postponement of quality abrupt decreases in some steps of error rates compared with *approximate memory*. The application fft has a similar quality in all error rates in AxRAM and *approximate memory*.

Most of the **memory-bound applications** have a similar quality behavior on AxRAM and *approximate memory* environments, with or without error rate steps offset. The memory-bound applications that do not suffer significant differences in quality with *AxRAM* are 2mm and floyd–warshall. Applications bzip2, bunzip2, and dijkstra have an offset of some error rates, depending on the threshold. Qsort achieves the highest improvements in quality with AxRAM among memory-bound applications with several error rates offset.

In general, our results show that crashes strongly influence the average output quality of application executions. Nevertheless, some applications show a different behavior between crash increases and quality depreciation. This evidences that crashes are not the only influence on the average output quality.

6.5. Energy

The quality analysis in Fig. 11 shows how AxRAM allows applications to produce higher quality results when subjected to a given error rate, which is analogous to produce higher quality results at a fixed energy budget. However, increasing the output quality also allows applications to be executed at higher error rates while still meeting a quality requirement. Thus, instead of increasing quality at an energy budget, AxRAM can also save energy for a given quality constraint. To evaluate this, we calculate the relative energy consumption taking into account a quality threshold to profile the application. The baseline can be called an accurate memory, defined as a memory that yields the very low probability of error of 1 in 10^{-12} operations. The profiling of the application on this environment statistically guarantees an average quality on a certain relative energy consumption. The accurate memory region of the protected stack is negligible compared to the energy consumption of the entire memory array. In our experiments, the size of the stack of all applications is at most 250 kB.

To find the relative energy consumption to an average quality, we associate each error rate with a respective dynamic energy consumption. The expected behavior without the influence of crashes is that energy consumption decreases smoothly as the error rate increases. Nonetheless, the growth of data and flow crashes causes abrupt decreases in energy consumption due to the termination of executions earlier than expected. Moreover, a large number of timeouts cause an increase in energy consumption.

Fig. 12 shows the geometric mean of the minimum required energy to achieve average quality thresholds from 95% to 100% at a 0.5% step. AxRAM achieves more energy savings than *approximate memory* at all quality thresholds. Nevertheless, depending on the quality threshold AxRAM does not exhibit energy gains to all applications compared to *approximate memory*, e.g. at a 95% quality threshold that AxRAM saves energy to 9 out of 12 applications with an 8.92% mean of less energy.

Some applications are not achieving 100% quality with both *approximate memory* and AxRAM. Thus, an accurate execution is needed to reach this quality, which causes a considerably increasing of energy consumption. Nevertheless, 100% quality represents an accurate output and a requisite to applications that execute in approximate environments is that some inaccuracy is tolerated.

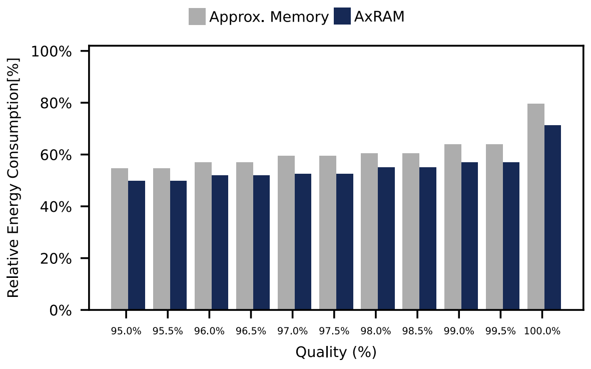


Fig. 12. Mean of relative energy to achieve average qualities thresholds.

6.6. Quality-energy efficiency

Quality metrics show how much the output deviates from the original result. Energy metrics show to what extent the approximation provides benefits. These two types of metrics show different aspects of the results. Thus, we define a combined metric that represents both aspects, the Quality-Energy Efficiency (QEE). This metric is defined by $\frac{Q}{E}$, where Q is the normalized quality and E is the percentage of energy relative to the consumption of an accurate memory. Fig. 13 shows the average QEE to all evaluated applications and error rates.

An accurate memory has QEE equals to 1.0 since the quality of its outputs is 100% and its relative energy consumption is 100%. Thus, the results of QEE less than 1.0 are inefficient due to being below the results of an accurate memory. All evaluated applications show some error rate with QEE higher than 1.0. AxRAM has a higher peak of QEE than *approximate memory* for 10 out of the 12 evaluated applications due to the postponement of the QEE fall to higher error rates. The QEE line of mandelbrot, reg_detect, and spectralnorm shows that these applications are not affected by memory errors with AxRAM protections.

Despite QEE represents a combined metric of quality and energy, it does not show individual values for quality or energy. A very low energy consumption may represent an increase of QEE even with low quality. Applications dijkstra and jpeg show this increase in the higher error rates with approximate memory, where the average energy consumption is very low, due to crashes at the beginning of the executions, but the average quality is almost null. Despite that, the peak of QEE for all applications, in our evaluated scenario, achieves quality higher than 90% for both AxRAM and *approximate memory*.

7. Discussion

AxRAM proposes a generic memory architecture that implements a set of approximate states, which are operating points that induce read and write errors in the stored data. By controlling the error rate and the region of the memory array that is affected, the AxRAM interface allows an external agent to control the degree of approximation provided, inducing energy savings by tolerating some quality depreciation.

In our simulated evaluation, however, we employ AxRAM in a limited scenario in which one single-threaded embedded application runs in bare metal in the CPU, with full control over the entire memory array and AxRAM control knobs. In this scenario, each induced memory error is a single random bit flip in the memory data word per operation. In this section, we discuss this simplified scenario and the implications of more realistic conditions in the proposed technique.

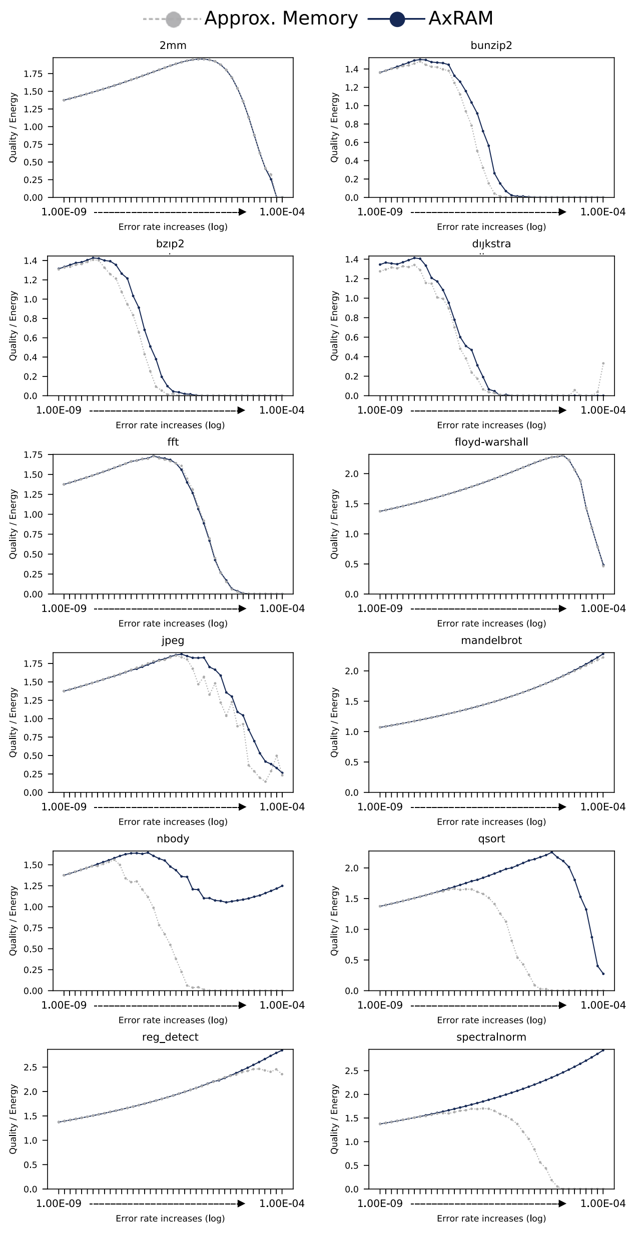


Fig. 13. Quality-energy efficiency for each application.

7.1. Approximation technique applicability

Billions of low-power devices are producing a new kind of data, where the exactness required to process traditional data is usually not necessary and errors are often tolerated [6]. These data include human perception contexts, like recognition and vision, and other types of processing, like machine learning and data mining. The approximate computing paradigm aims to exploit degrees of error tolerance while maintaining acceptable results. AxRAM explores approximations at the memory level and raises the likelihood of acceptable quality to achieve energy gains.

In our evaluation scenario, we target applications or kernels that repeatedly execute over many different inputs, e.g. learning inferences over images acquired from a camera. In these applications, it is possible to recover from lower than tolerable quality outputs by re-executing or discarding useless outputs. Nevertheless, AxRAM is applicable to scenarios of data approximation that could be benefited from its protections.

The applicability of AxRAM is the same as the general approximate computing techniques, entirely dependent on the context of the applications. The data that tolerate errors should be substantial to achieve the benefits that AxRAM provides. The implementation of critical data isolation with stack protection shows potential gains, but do not protect critical data that rely on specific data structures, as the linked lists used on dijkstra. The use of the stack also should be relatively small compared to the entire memory to use this implementation. Applications with overmuch recursive calls may increase the accurate part of memory, which may hold the energy gains of AxRAM.

7.2. Quality control in production scenarios

In production environments, applications accept some quality loss in approximate environments, but with a tolerable limit. Thus, it is necessary to find an approximation level that respects this limit. To calculate the relative energy savings of our technique, we consider an average quality threshold achieved in each error rate, where it is possible to obtain lower than required quality in individual executions. However, we ensure through profiling that, on average, the application kernel will meet a required quality target obtained.

After deployment, the output quality cannot be computed for every execution instance. Nevertheless, we get in a training phase the error rate to achieve each average quality threshold. Since AxRAM eliminates data crashes and quality has already been taken into account in the training phase, a watchdog technique can be employed to avoid application stalling (timeout crashes), leading quality and energy to converge to the observed in the training phase. In case a more dynamic evaluation is required, some execution instances may be sampled and elected for non-approximate computation, fine-tuning the operating point selection.

7.3. Multi-bit memory errors

Our simulated evaluation scenario considers single-bit soft errors to simplify error modeling. The effects and behavior of AxRAM under single-bit errors are similar to those of multi-bit errors. Incorrect pointers resulting from multi-bit errors would also be truncated by the AxRAM addressing mask that would avoid data crashes caused by these pointers. The correction provided by this mask would achieve similar results on both scenarios, with the same level of approximation and order of magnitude on single and multiple-bit errors. Evidence from Sangchoolie et al. [54] suggests that the single bit-flip model is enough in resilience studies for less pessimistic fault injection scenarios since the outcomes are similar in most cases.

7.4. Implications of approximation control

The AxRAM interface exposes to the environment – the application, in the simpler embedded system scenario, an application-level library, the OS, or the middleware – control knobs in the form of memory-mapped registers. Although these registers can be written at any point at execution time, changing the approximate state or the approximated memory banks is a potentially time-consuming operation, similar to changing power states or dynamic voltage–frequency scaling (DVFS). For this reason, it is desirable to perform coarse-grained control of approximations, reducing state changes to a minimum, such as in the evaluated scenarios, where we set the approximate region and state at application kernel start-up and keep the setting until the end.

8. Conclusion

AxRAM is an approximate data access interface that avoids execution crashes implicitly and without significant performance and energy costs. AxRAM has two main features that increase execution resilience: an addressing scheme to treat incorrect memory references and critical data protection of the system stack. Incorrect memory references are the main cause of a great part of the execution crashes, and AxRAM proposes three techniques to treat these data: ignoring out-of-bounds pointers, storing zero in the destination register, and truncating these values into allowed memory bounds. The protection of the system stack provides the error isolation of critical data to many kinds of applications without user intervention, such as function pointers, local variables, and control indexes.

Our experimental evaluation shows that the critical data implicit protection decreases the number of flow crashes, while the addressing schemes are able to avoid data crashes. An implementation of AxRAM in an embedded computing scenario featuring a dual-voltage SRAM memory with configurable reliability shows a reduction of 51% of execution crashes across all error rates compared to an unprotected approximate memory. At 95% average output quality, AxRAM shows dynamic energy savings of 9% and a higher peak of quality-energy efficiency for 10 out of 12 applications when compared to an unprotected approximate memory. For the same 95% average output quality, when compared to a system with non-approximate memory producing exact results, AxRAM reduces dynamic energy consumption in half.

AxRAM improves execution resilience using techniques applicable to common types of applications. However, some critical data are application-specific and are not covered by this protection. Thus, as future work, we intend to identify, in the application, common structures and code patterns that lead to execution crashes if exposed to errors.

CRediT authorship contribution statement

João Fabrício Filho: Conceptualization, Methodology, Software, Investigation, Writing - original draft, Visualization. **Isaías B. Felzmann:** Conceptualization, Methodology, Software, Investigation, Writing - original draft, Visualization. **Rodolfo Azevedo:** Conceptualization, Validation, Writing - review & editing, Supervision. **Lucas F. Wanner:** Conceptualization, Validation, Writing - review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the São Paulo Research Foundation (FAPESP) grant #2018/24177-0; the National Council for Scientific and Technological Development — Brazil (CNPq) grants #404261/2016-7 and #438445/2018-0; and the Coordination for the Improvement of Higher Education Personnel — Brazil (CAPES) – Finance Code 001, PROCAD#2966/2014. Figures 2, 3, 8—11 © 2019 IEEE Reprinted, with permission, from [31].

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, IEEE Micro 32 (3) (2012) 122–134, http://dx.doi.org/10.1109/MM.2012.17.

[2] A. Aponte-Moreno, A. Moncada, F. Restrepo-Calle, C. Pedraza, A review of approximate computing techniques towards fault mitigation in HW/SW systems, in: 2018 IEEE 19th Latin-American Test Symposium, LATS, IEEE, 2018, pp. 1–6, http://dx.doi.org/10.1109/LATW.2018.8347241.

[3] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, P. Gupta, Measuring the impact of memory errors on application performance, IEEE Comput. Archit. Lett. 16 (1) (2017) 51–55, http://dx.doi.org/10.1109/LCA.2016.2599513.

[4] X. Li, D. Yeung, Application-level correctness and its impact on fault tolerance, in: Proceedings - International Symposium on High-Performance Computer Architecture, IEEE, 2007, pp. 181–192, http://dx.doi.org/10.1109/HPCA.2007.346196.

[5] Y. Chen, J. Chhugani, P. Dubey, C.J. Hughes, D. Kim, S. Kumar, V.W. Lee, A.D. Nguyen, M. Smelyanskiy, Convergence of recognition, mining, and synthesis workloads and its implications, Proc. IEEE 96 (5) (2008) 790–807, http://dx.doi.org/10.1109/JPROC.2008.917729.

[6] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D.A. Prener, S. Shukla, V. Srinivasan, Z. Sura, Approximate computing: Challenges and opportunities, in: 2016 IEEE International Conference on Rebooting Computing, ICRC 2016 - Conference Proceedings, IEEE, 2016, pp. 1–8, http://dx.doi.org/10.1109/ICRC.2016.7738674.

[7] T. Moreau, J. San Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. Enright Jerger, A. Sampson, A taxonomy of general purpose approximate computing techniques, IEEE Embedded Syst. Lett. 10 (1) (2018) 2–5, http://dx.doi.org/10.1109/LES.2017.2758679.

[8] M.A. Ben Khadra, An introduction to approximate computing, 2017, CoRR abs/1711.06115, arXiv:1711.06115.

[9] P. Loloeyan, H. Nikmehr, Do we need approximate-aware cache in dark silicon era? in: International Congress on Technology, Communication and Knowledge, ICTCK, IEEE, 2016, pp. 466–472, http://dx.doi.org/10.1109/ICTCK.2015.7582713.

[10] J. Liu, B. Jaiyen, R. Veras, O. Mutlu, RAIDR: Retention-aware intelligent DRAM refresh, in: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1–12, http://dx.doi.org/10.1145/2366231.2337161.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, Architecture support for disciplined approximate programming, in: International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, 2012, pp. 301–312, http://dx.doi.org/10.1145/2150976.2151008.

[12] V. De, S. Vangal, R. Krishnamurthy, Near threshold voltage (NTV) computing: Computing in the dark silicon era, IEEE Des. Test 34 (2) (2017) 24–30, http://dx.doi.org/10.1109/MDAT.2016.2573593.

[13] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, D. Grossman, EnerJ: Approximate data types for safe and general low-power computation, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, New York, New York, USA, 2011, p. 164, http://dx.doi.org/10.1145/1993498.1993518.

[14] I. Felzmann, J. Fabrício Filho, R. Azevedo, L. Wanner, Impact of memory approximation on energy efficiency, in: Proceedings of the Symposium on High-Performance Computing Systems, WSCAD 2018, Institute of Electrical and Electronics Engineers Inc., 2018, pp. 53–60, http://dx.doi.org/10.1109/WSCAD.2018.00018.

[15] M. Gottscho, A. BanaiyanMofrad, N. Dutt, A. Nicolau, P. Gupta, DPCS: Dynamic power/capacity scaling for SRAM caches in the nanoscale era, ACM Trans. Archit. Code Optim. 12 (3) (2015) 1–26, http://dx.doi.org/10.1145/2792982.

[16] A. Sampson, J. Nelson, K. Strauss, L. Ceze, Approximate storage in solid-state memories, in: IEEE/ACM International Symposium on Microarchitecture, MICRO, ACM Press, New York, New York, USA, 2013, pp. 25–36, http://dx.doi.org/10.1145/2540708.2540712.

[17] M. Carbin, S. Misailovic, M.C. Rinard, M. Carbin, S. Misailovic, M.C. Rinard, Verifying quantitative reliability for programs that execute on unreliable hardware, ACM SIGPLAN Not. 48 (10) (2013) 33–52, http://dx.doi.org/10.1145/2544173.2509546.

[18] J. Lucas, M. Alvarez-Mesa, M. Andersch, B. Juurlink, Sparkk: Quality-scalable approximate storage in DRAM, in: The Memory Forum, 2014, pp. 1–6.

[19] S. Venkataramani, V.K. Chippa, S.T. Chakradhar, K. Roy, A. Raghunathan, Quality programmable vector processors for approximate computing, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, ACM, New York, NY, USA, 2013, pp. 1–12, http://dx.doi.org/10.1145/2540708.2540710.

[20] B. Boston, A. Sampson, D. Grossman, L. Ceze, Probability type inference for flexible approximate programming, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vol. 50, OOPSLA 2015, ACM Press, New York, New York, USA, 2015, pp. 470–487, http://dx.doi.org/10.1145/2814270.2814301.

[21] M. Samadi, D.A. Jamshidi, J. Lee, S. Mahlke, Paraprox: pattern-based approximation for data parallel applications, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 49, ASPLOS '14, ACM Press, New York, New York, USA, 2014, pp. 35–50, http://dx.doi.org/10.1145/2541940.2541948.

[22] S. Campanoni, G. Holloway, G.-Y. Wei, D. Brooks, HELIX-UP: Relaxing program semantics to unleash parallelization, in: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 235–245.

[23] H. De Silva, A.E. Santosa, N.-M. Ho, W.-F. Wong, ApproxSymate: path sensitive program approximation using symbolic execution, in: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, ACM Press, New York, New York, USA, 2019, pp. 148–162, http://dx.doi.org/10.1145/3316482.3326341.

[24] M. Cohen, H.S. Zhu, S.E. Emgin, Y.D. Liu, Energy types, in: ACM SIGPLAN Notices, ACM Press, New York, New York, USA, 2012, pp. 831–849, http://dx.doi.org/10.1145/2398857.2384676.

[25] M. de Kruijf, S. Nomura, K. Sankaralingam, Relax: An architectural framework for software recovery of hardware faults, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, ACM, New York, NY, USA, 2010, pp. 497–508, http://dx.doi.org/10.1145/1815961.1816026.

[26] A. Ranjan, S. Venkataramani, Z. Pajouhi, R. Venkatesan, K. Roy, A. Raghunathan, STAxCache: An approximate, energy efficient STT-MRAM cache, in: Conference on Design, Automation & Test in Europe, DATE, IEEE, 2017, pp. 356–361, http://dx.doi.org/10.23919/DATE.2017.7927016.

[27] M. Samadi, J. Lee, D.A. Jamshidi, A. Hormati, S. Mahlke, SAGE: self-tuning approximation for graphics engines, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-4, ACM Press, New York, New York, USA, 2013, pp. 13–24, http://dx.doi.org/10.1145/2540708.2540711.

[28] D.S. Khudia, B. Zamirai, M. Samadi, S. Mahlke, D.S. Khudia, B. Zamirai, M. Samadi, S. Mahlke, Rumba: an online quality management system for approximate computing, ACM SIGARCH Comput. Archit. News 43 (3) (2015) 554–566, http://dx.doi.org/10.1145/2872887.2750371.

[29] Y. Wang, J. Deng, Y. Fang, H. Li, X. Li, Resilience-aware frequency tuning for neural-network-based approximate computing chips, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 25 (10) (2017) 2736–2748, http://dx.doi.org/10.1109/TVLSI.2017.2682885.

[30] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, D. Grossman, Monitoring and debugging the quality of results in approximate programs, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, ACM, New York, NY, USA, 2015, pp. 399–411, http://dx.doi.org/10.1145/2694344.2694365.

[31] J. Fabrício Filho, I. Felzmann, R. Azevedo, L. Wanner, A resilient interface for approximate data access, in: Proceedings of the IX Brazilian Symposium on Computing Systems Engineering, SBESC'19, IEEE, 2019, pp. 1–8, http://dx.doi.org/10.1109/SBESC49506.2019.9046069.

[32] L. Kugler, Is "good enough" computing good enough? Commun. ACM 58 (5) (2015) 12–14, http://dx.doi.org/10.1145/2742482.

[33] A. Raha, S. Sutar, H. Jayakumar, V. Raghunathan, Quality configurable approximate DRAM, IEEE Trans. Comput. 66 (7) (2017) 1172–1187, http://dx.doi.org/10.1109/TC.2016.2640296.

[34] F. Samie, L. Bauer, J. Henkel, An approximate compressor for wearable biomedical healthcare monitoring systems, in: International Conference on Hardware/Software Codesign and System Synthesis Companion - (CODES+ISSS), IEEE, 2015, pp. 133–142, http://dx.doi.org/10.1109/CODESISSS.2015.7331376.

[35] T. Moreau, F. Augusto, P. Howe, A. Alaghi, L. Ceze, Exploiting quality-energy tradeoffs with arbitrary quantization, in: International Conference on Hardware/Software Codesign and System Synthesis Companion - (CODES+ISSS), ACM Press, New York, New York, USA, 2017, pp. 1–2, http://dx.doi.org/10.1145/3125502.3125544.

[36] D. Burke, D. Jenkus, I. Qiqieh, R. Shafik, S. Das, A. Yakovlev, Significance-driven adaptive approximate computing for energy-efficient image processing applications, in: International Conference on Hardware/Software Codesign and System Synthesis Companion - (CODES+ISSS), ACM Press, New York, New York, USA, 2017, pp. 1–2, http://dx.doi.org/10.1145/3125502.3125554.

[37] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, J. Henkel, Invited - Cross-layer approximate computing: from logic to architectures, in: Design Automation Conference, DAC, ACM Press, New York, New York, USA, 2016, pp. 1–6, http://dx.doi.org/10.1145/2897937.2906199.

[38] J. Wang, B.H. Calhoun, Minimum supply voltage and yield estimation for large SRAMs under parametric variations, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 19 (11) (2011) 2120–2125, http://dx.doi.org/10.1109/TVLSI.2010.2071890.

[39] M. Shoushtari, A. BanaiyanMofrad, N. Dutt, Exploiting partially-forgetful memories for approximate computing, IEEE Embedded Syst. Lett. 7 (1) (2015) 19–22, http://dx.doi.org/10.1109/LES.2015.2393860.

[40] J.S. Miguel, J. Albericio, A. Moshovos, N.E. Jerger, Doppelgänger: A cache for approximate computing, in: International Symposium on Microarchitecture, MICRO, ACM Press, New York, New York, USA, 2015, pp. 50–61, http://dx.doi.org/10.1145/2830772.2830790.

[41] F. Frustaci, D. Blaauw, D. Sylvester, M. Alioto, Approximate SRAMs with dynamic energy-quality management, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 24 (6) (2016) 2128–2141, http://dx.doi.org/10.1109/TVLSI.2015.2503733.

[42] V.K. Chippa, D. Mohapatra, K. Roy, S.T. Chakradhar, A. Raghunathan, Scalable effort hardware design, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 22 (9) (2014) 2004–2016, http://dx.doi.org/10.1109/TVLSI.2013.2276759.

[43] Y. Verdeja Herms, Y. Li, Crash skipping: A minimal-cost framework for efficient error recovery in approximate computing environments, in: Proceedings of the 2019 on Great Lakes Symposium on VLSI, GLSVLSI '19, ACM Press, New York, New York, USA, 2019, pp. 129–134, http://dx.doi.org/10.1145/3299874.3317986.

[44] P. Roy, R. Ray, C. Wang, W.F. Wong, ASAC: Automatic sensitivity analysis for approximate computing, in: ACM SIGPLAN Notices, ACM Press, New York, New York, USA, 2014, pp. 95–104, http://dx.doi.org/10.1145/2597809.2597812.

[45] M. Powell, Se-Hyun Yang, B. Falsafi, K. Roy, T.N. Vijaykumar, Gated-V/sub dd/: a circuit technique to reduce leakage in deep-submicron cache memories, in: ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514), 2000, pp. 90–95, http://dx.doi.org/10.1109/LPE.2000.155259.

[46] I.B. Felzmann, M.M. Susin, L. Duenha, R. Azevedo, L.F. Wanner, ADeLe: A description language for approximate hardware, Future Gener. Comput. Syst. 102 (2020) 245–258, http://dx.doi.org/10.1016/J.FUTURE.2019.07.073.

[47] S. Rigo, G. Araújo, M. Bartholomeu, R. Azevedo, ArchC: A systemC-based architecture description language, in: Proceedings - Symposium on Computer Architecture and High Performance Computing, IEEE, 2004, pp. 66–73, http://dx.doi.org/10.1109/SBAC-PAD.2004.8.

[48] A.N. Avanaki, Exact global histogram specification optimized for structural similarity, Opt. Rev. 16 (6) (2009) 613–621, http://dx.doi.org/10.1007/s10043-009-0119-z, arXiv:0901.0065v1.

[49] L.-N. Pouchet, Polybench/C: The polyhedral benchmark suite, 2012, URL http://web.cs.ucla.edu/~pouchet/software/polybench/.

[50] I. Gouy, The computer language benchmarks game, 2004, URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[51] G. Fursin, Collective benchmark — enabling realistic benchmarking and optimization, 2008, URL http://ctuning.org/cbench.

[52] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: IEEE International Workshop on Workload Characterization, WWC, IEEE, 2001, pp. 3–14, http://dx.doi.org/10.1109/WWC.2001.990739.

[53] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, P. Lotfi-Kamran, AxBench: A multiplatform benchmark suite for approximate computing, IEEE Des. Test 34 (2) (2017) 60–68, http://dx.doi.org/10.1109/MDAT.2016.2630270.

[54] B. Sangchoolie, K. Pattabiraman, J. Karlsson, One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors, in: IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, 2017, pp. 97–108, http://dx.doi.org/10.1109/DSN.2017.30.

João Fabrício Filho received his B.S. degree in Informatics (2015) and his M.S. degree in Computer Science (2017) from the State University of Maringá (UEM). He is currently a Ph.D. student at the University of Campinas (UNICAMP) and works at the Federal University of Technology — Paraná (UTFPR) Campus Campo Mourão.

Isaías Bittencourt Felzmann received the B.Eng. degree (2015) from the Federal University of Santa Maria and M.S. degree (2019) from the University of Campinas. He is currently a Ph.D. student at the University of Campinas, working on architecture-level integration and evaluation of Approximate Computing.

Rodolfo Azevedo is an associate professor at University of Campinas (UNICAMP). He received his Ph.D. in Computer Science from UNICAMP in 2002 and is a member of the Computer Science graduate program where he advises master and Ph.D. students. He got four best papers in conferences (SBAC-PAD 2004, SBAC-PAD 2008, WSCAD-SSC 2012 and SBAC-PAD 2018). In 2012 he received the Zeferino Vaz Academic Award and the newly created UNICAMP Teaching Award. He has had a CNPq Research Fellowship since 2006. In the last 10 years, he has been honored 6 times in the Computer Science and Computer Engineering graduations.

Lucas Wanner received the B.S. (2004) and M.S. (2006) degrees from the Federal University of Santa Catarina, and the Ph.D. (2014) from the University of California, Los Angeles. He joined the faculty of the Institute of Computing at the University of Campinas in 2015, where he is currently an Assistant Professor in the Computer Systems Department. His research focuses on energy efficiency in hardware and software.