AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing

Isaías Felzmann*, João Fabrício Filho*†, Lucas Wanner*

*Institute of Computing, University of Campinas, Brazil
†Federal University of Technology - Paraná - Campus Campo Mourão, Brazil
isaias.felzmann@ic.unicamp.br, joaof@utfpr.edu.br, lucas@ic.unicamp.br

***Abstract*—Representing the interaction between accurate and approximate hardware modules at the architecture level is essential to understand the impact of Approximate Computing in a general-purpose computing scenario. However, extensive effort is required to model approximations into a baseline instruction-level simulator and collect its execution metrics. In this work, we present the AxPIKE ISA simulation environment, a tool that allows designers to inject models of hardware approximation at the instruction level and evaluate their impact on the quality of results. AxPIKE embeds a high-level representation of a RISC-V system and produces a dedicated control mechanism, that allows the simulated software to manage the approximate behavior of compatible execution scenarios. The environment also provides detailed execution statistics that are forwarded to dedicated tools for energy accounting. We apply the AxPIKE environment to inject integer multiplication and memory access approximations into different applications and demonstrate how the generated statistics are translated into energy-quality trade-offs.**

***Index Terms*—Approximate Computing, Architectural simulation, Error injection**

I. Introduction

Approximate Computing has introduced hardware modules that allow increased energy efficiency on computation in exchange for relaxed quality requirements of the final output. Common hardware techniques to achieve some level of approximation are the design of dedicated logic that outputs error-prone results, the scaling of external parameters that have side-effects on quality, and the inclusion of configuration knobs that introduce deviations [1]. These hardware modules are often evaluated in isolation from the rest of the system or in applications by replacing some high-level software operation with a specific software model. This approach tends to disregard the interaction and integration between the approximation units and other components in the system, limiting essential co-design possibilities [2].

An alternative for evaluation is to inject the approximations at the instruction level. Thus, instead of, for example, evaluating a hardware multiplier using a subset of random operands or overloading the high-level multiplication operator with a specialized function [3], the designer would replace the behavior multiplication instructions in an ISA simulator. This places the approximation where it would be noticed in the application, considering the interaction with instructions, and reducing the software overhead of higher-level modeling.

Setting up a simulator for this purpose, however, is not a trivial task. CPU simulators are designed to produce the application output and some simplified aggregated execution statistics, such as a global instruction counter, number of memory accesses, and cache misses. Even though these statistics can be used to estimate performance, more data is needed to account for energy. Moreover, although simulators often disclose the modeling of instructions, allowing for straightforward changes, control mechanisms such as instruction issue and decode, memory and register accesses, and the production of statistics are often embedded in the software structure, requiring extensive effort to understand and modify the simulator.

In this paper, we present AxPIKE, a RISC-V-based simulation environment for the evaluation of approximate hardware modules. AxPIKE uses a CPU simulator to inject approximation into instructions or data accesses. Also, an ISA-level interface and a software library control the approximate behavior, protecting critical regions to tweak the energy-quality trade-off. The approximate behavior is represented in ad-hoc instruction-level models interpreted through a configuration file. The CPU simulator produces the application output and detailed execution statistics to account for energy, such as memory access traces, instruction-specific counters, and specific operand traces. In the AxPIKE environment, these statistics are forwarded to dedicated tools and models that evaluate energy and quality.

We demonstrate the AxPIKE environment simulating some typical Approximate Computing scenarios. For a set of applications, we inject an approximate multiplication model [3] and a DRAM error model based on voltage overscaling [4]. We configure the simulator to produce statistics about executed multiplications and DRAM accesses and then apply specialized tools to estimate energy consumption. Our results show how AxPIKE enhances the evaluation of approximation scenarios.

II. Related Work

Approximate Computing designers have proposed different solutions to evaluate approximations in the architecture level. VarEMU [5] extends instructions by replacing or augmenting them with custom software models, estimating energy based on internal power models. It does, however, limit the representation of microarchitecture details to improve performance, limiting

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.

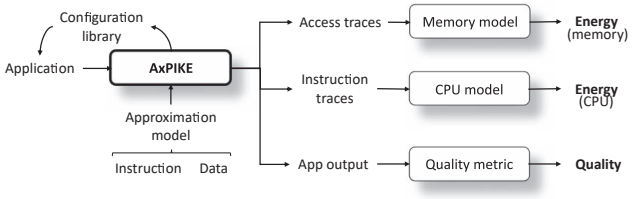

Fig. 1. Workflow of the AxPIKE simulation environment. *It produces detailed execution statistics to estimate quality and energy of application execution.*

```
1  IM ILM_EA(word a, word b, word r, processor_t* p) {
2    // r ≈ a * b
3    Reinterpret(a = m1 + q1, where m1 = 2^k1);
4    Reinterpret(b = m2 + q2, where m2 = 2^k2);
5    r = 2^(k1+k2) + q2*2^k1 + q1*2^k2;
6  }
7
8  DM AxRAM(processor_t* p, source_t* source, void* data) {
9    if (source->hierarchy == DRAM &&
10       p->get_state()->prv == USER) {
11       UniformBitFlip(1.4E-5, data);
12   }
13 }
```

Fig. 2. Sample AxPIKE approximation models. *They represent instruction behavior (IM) or changes in data accesses (DM).*

the accuracy of execution statistics to the modeled details and restricting the evaluation of low-level approximation. Also embedding its own power model, React [6] instruments applications at design time for later simulation. Apart from requiring changes in the source code and recompilation, it limits the evaluation to pre-defined approximation techniques and statistics, not allowing further configuration.

Approxilyzer [7] evaluates the impact of perturbations in the execution caused by Approximate Computing using single bit-flip soft-error models. The approach allows designers to estimate the application resiliency to errors but limits a more accurate representation of many approximation techniques. ADeLe [8] introduces customizable error and power models to be applied to a custom CPU model. Although stacking levels of customization enlargens the frontiers of design space exploration, it tends to increase the effort on setting up the evaluation. Moreover, the simulation is only as accurate as the microarchitecture is detailed in the base CPU model, creating a direct dependency on the representativeness of results.

In this work, AxPIKE inherits the customization of error models from VarEMU [5] and ADeLe [8], while linking it with the rapid setup time of React [6] and Approxilyzer [7]. Based on the Spike RISC-V ISA simulator, off-the-shelf AxPIKE can run an application in a multi-privilege multicore representation of a CPU including ISA-level microarchitectural details, while still allowing injection of errors and collection of statistics. Also, we refrain from embedding power models in favor of producing data for external energy estimation. Thus, AxPIKE leaves to the CPU simulator only the aspects that are related to the behavioral execution of instructions itself, while still allowing flexible extension and collecting enough information for the evaluation of Approximate Computing designs.

III. AxPIKE Simulation Environment

AxPIKE core simulator is forked from the Spike RISC-V ISA simulator. Figure 1 shows the workflow to evaluate an application. The core simulator is compliant with the RISC-V ISA and runs binaries compiled using standard tools. A high-level descriptive file configures the simulation and associates instructions and data accesses with custom approximation models [8]. A priori, AxPIKE does not require any changes in the application to inject approximations. However, in a common scenario in which certain structures or code regions need to be protected, we provide a low-overhead configuration interface based on dedicated control registers [9]. For convenience, AxPIKE generates a software library for the current configuration of approximations, translating comprehensive function-calls into the lower-level register-based control.

The execution of AxPIKE produces the application output and simulation statistics. The output is evaluated by a scenario-specific quality metric to determine the quality of results. By default, the statistics include memory accesses and instruction counters, such as the number of accesses and misses for each level of cache and number of executions of each instruction, all detailed according to privilege level, approximation configuration, and region in the application code. Additionally, AxPIKE provides a configurable statistics generator to produce custom traces, as well as other aggregated performance metrics.

A. Approximation Modeling and Injection

In AxPIKE, approximation hardware is modeled as functions that can be injected into any instruction in the ISA or modify the data read from or written to any register or memory location, including caches and the main memory. These models can receive user-defined parameters and access to the full state of the simulated CPU. Moreover, the data models provide metadata about the operation, such as source device, the type of operation (read, write, instruction fetch), virtual and physical addresses, and whether there was a TLB entry for it. Thus, the models allow a flexible and extensible instruction-level instrumentation of the approximated scenario.

Figure 2 exemplifies the approximation models to implement an approximate logarithm multiplier (*ILM_EA* [3]) and to inject errors in data read from the DRAM main memory (*AxRAM* [4]). The *word* datatype holds references to custom parameters that assume the size of a data word in the simulated CPU. In this multiplier scenario, they will refer to the respective registers in the register bank. The DRAM approximation emulates a scenario in which the supply voltage of the DRAM is scaled below common guardbands, subjecting the data to errors modeled as uniformly distributed bitflips. Supervisor and machine privilege code (e.g. the Operating System) run in protected memory regions and are thus exempt from these errors.

To effectively associate the models with instructions and data accesses, AxPIKE uses a configuration file [8]. Figure 3 exemplifies the configuration to inject the multiplier and the DRAM models. The configuration begins declaring the models, in which the metadata (`processor_t* p`, `source_t* source`, and `void* data`) are implicitly assumed and referenced when processing the configuration. The multiplier

```
IM ILM_EA(word a, word b, word r);
DM AxRAM();

approximation Approx_ILM_EA {
  instruction mul {
    alt_behavior = ILM_EA(FETCH_RS1, FETCH_RS2, FETCH_RD);
  }
}

approximation Approx_AxRAM {
  mem_read = AxRAM();
}
```

Fig. 3. Sample AxPIKE configuration. *Approximation models are distributed into approximation states that can be switched off and on dynamically.*

model replaces the behavior of the multiplication (*mul*) instruction (*alt(ernative)_behavior*) and receives the multiplication operands. Alternatively, a model could be injected before (*pre_behavior*) or after (*post_behavior*) the instruction execution. The DRAM model is injected when reading data from memory (*mem_read*), regardless of the instruction. Similarly, the error could affect write accesses (*mem_write*) and other storage structures such as the register bank (*regbank_[read/write]*).

The models are grouped together into higher-level *approximations*. An approximation is an entity that defines a state of the simulated system, which behavior can be described by multiple models injected into multiple instructions. For example, an approximate multiplier hardware could also affect multiply-and-accumulate instructions, in which case the software model differs but would be grouped in the same *approximation*. Approximations can be enabled or disabled at execution time, defining when the models should be injected or not.

B. Software Control Interface

AxPIKE allows the simulated software to take control over the approximation scenario using a control interface based on Control and Status Registers. The registers inform the software about which approximation states are available and offer a configuration knob for them to be activated or deactivated [9]. Figure 4 shows an example in which the main computation kernel of a matrix multiply application will use the *ILM_EA* approximate logarithm multiplier [3]. The interface library is automatically generated by the simulator and defines the functions to activate and deactivate the approximations. Approximations are activated by setting the corresponding bits in the `urkst` control register, that defines their status for any code running in user privilege level. Similarly, the interface allows approximations to be independently controlled by machine and supervisor code, and the state can be controlled by a supervisor system, waiving changes to the user-level code.

C. Statistics generator

To support performance and energy evaluation, AxPIKE provides detailed data about the execution. A configurable statistics generator allows designers to analyze such data. This allows, for example, the creation of custom performance counters and traces, such as detailed memory traces to feed external tools for energy estimation and logging operands and results of every instruction to produce activity factors for RTL simulation. Collecting statistics represents a significant

axpike_iface.h :

```
#define Approx_ILM_EA 1
#define Approx_AxRAM  2

#define axpike_activate(approx) \
  asm volatile ("csrrs %0, urkst, %1" :: "rK"(approx));

#define axpike_deactivate(approx) \
  asm volatile ("csrrc %0, urkst, %1" :: "rK"(approx));
```

matrix_multiply.c :

```
#include "axpike_iface.h"
#define SIZE 100

int main(int argc, char* argv[]) {
  int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
  int i, j, k;

  read_inputs(A, B);

  axpike_activate(Approx_ILM_EA);

  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      C[i][j] = 0;
      for (k = 0; k < SIZE; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }

  axpike_deactivate(Approx_ILM_EA);

  write_output(C);

  return 0;
}
```

Fig. 4. AxPIKE control interface. *Software communicates with the simulator using control registers via a dedicated software library.*

overhead in an AxPIKE execution. For example, for the simple Matrix Multiply application represented in Figure 4, tracing every memory access almost doubles the execution time and produces a log file of more than 40X the size, when compared to tracing only accesses in the main DRAM memory.

Even when the customizable statistics generation class is not used, AxPIKE produces some aggregated instruction and memory access counters by default. Table I exemplifies the default counters for the Matrix Multiplication application. The instruction counters are detailed for each instruction in the ISA, and for each level in the memory hierarchy in the case of the access counters. They also provide information on which was the privilege level of the instruction/access execution, as well as the active approximations at the time. In the example scenario, some of the multiplication instructions are executed in the approximate logarithm multiplier [3], while all memory accesses are subjected to the DRAM error model, even though some access may point to protected memory regions [4].

IV. Experimental Demonstration

To demonstrate AxPIKE in operation, we selected the applications Matrix multiply, Sobel, and JPEG to evaluate the approximate logarithm multiplier [3] and approximate DRAM access [4] scenarios. Both scenarios are exclusive. Matrix multiply operates on two 100x100 integer matrices, Sobel detects edges in a 256x256 image, and JPEG compresses a 512x512 bitmap. For each application, we compute the quality of results

TABLE I
STATISTICS COLLECTED BY AXPIKE EXECUTION FOR THE MATRIX MULTIPLY APPLICATION

Instruction statistics			
Instruction	**Prv.**[1]	**Approximation**	**Counter**
– All –	M/S	None	52,100,132
		Approx_ILM_EA	21,383
	U	None	21,036,963
		Approx_ILM_EA	7,082,321
Multiplication[2]	M/S	None	0
		Approx_ILM_EA	0
	U	None	93,143
		Approx_ILM_EA	1,000,000
Memory statistics			
Access level	**Prv.**[1]	**Approximation**	**Counter**
L1 cache	M/S	Approx_AxRAM	1,451,651
	U	Approx_AxRAM	8,373,444
L2 cache	M/S	Approx_AxRAM	52,308
	U	Approx_AxRAM	244,016
DRAM	M/S	Approx_AxRAM	51,540
	U	Approx_AxRAM	195,925

[1] Privilege: M = machine, S = supervisor, U = user
[2] Includes the whole integer multiplication family

TABLE II
QUALITY AND ENERGY EVALUATION OF SELECTED APPLICATIONS SUBJECTED TO APPROXIMATION

	ILM_EA		**AxRAM**	
Application	Quality	Energy	Quality	Energy
Matrix multiply	79.85%	99.62%	95.17%	90.46%
Sobel	79.55%	98.30%	95.12%	89.58%
JPEG	97.68%	99.62%	85.83%	88.53%

and estimate the energy relative to an execution that does not employ approximation. In our experiments, the applications are compiled as Linux binaries and the OS behavior is emulated by the RISC-V Proxy Kernel, running on a single-core RV64g CPU, configured with independent 32 KB instruction and data L1 caches and a single 128 KB L2 cache.

Table II summarizes quality and energy results. The quality metrics are the Mean Relative Error for Matrix multiply and the Structural Similarity Index for Sobel and JPEG. To compute DRAM energy, we configured AxPIKE to produce traces of every DRAM access, which were then forwarded to Ramulator [10] and DRAMPower [11] for estimation. The DRAM energy and error models were derived from experimental data from Chang *et al.* [12]. For the CPU energy, we consider the upper-bound savings for the multiplication operator, considering that the multiplier represents 8.5% of the energy cost of an instruction for a general RISC-V workload [13] and the instruction counters produced by AxPIKE. We consider that both the CPU and the DRAM contribute equally to the overall system energy cost. In the multiplication scenario, only the main computation kernel was affected by the approximation, because multiplications are part of application initialization and I/O phases, and these are critical to prevent application crashes. Thus, energy savings with the approximate multiplier are restricted. In the DRAM scenario, on the other hand, the entire application can be subjected to approximation, except for some small protected memory regions that store mostly control structures. Thus, approximating the memory accesses shows a wider margin to improve energy efficiency.

To produce simulation statistics and inject the approximations, AxPIKE imposes overhead in the base Spike simulator. Table III shows the execution time for each of the applications

TABLE III
COMPARISON OF SIMULATION PERFORMANCE UNDER DIFFERENT CONFIGURATIONS

Application	**Spike**	**AxPIKE**	**ILM_EA**	**AxRAM**
Matrix multiply	22 s	61 s	61 s	63 s
Sobel	8 s	23 s	23 s	23 s
JPEG	50 s	158 s	160 s	158 s

in the original Spike simulator, AxPIKE without injecting any approximation nor producing additional statistics, and each of the approximation scenarios. Although AxPIKE required three times longer to compute results in the experimented scenarios, it allows a richer evaluation allowing approximation control and producing extended statistics.

V. CONCLUSION

This work presented AxPIKE, an instruction-level simulation environment for the evaluation of Approximate Computing hardware designs. AxPIKE assists in injecting custom software models of the designs into a CPU simulator, taking advantage of the architectural-level implementation to produce detailed execution statistics. Thus, AxPIKE leaves to the CPU behavioral simulator only the task of behaviorally execute the target application, while producing detailed counters and traces for energy estimation. AxPIKE is free software and is available for download at `varchc.github.io/axpike`.

REFERENCES

[1] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, 2016.
[2] W. Liu, F. Lombardi, and M. Shulte, "A Retrospective and Prospective View of Approximate Computing [Point of View]," *Proceedings of the IEEE*, vol. 108, no. 3, pp. 394–399, 2020.
[3] M. S. Ansari, B. F. Cockburn, and J. Han, "A Hardware-Efficient Logarithmic Multiplier with Improved Accuracy," in *DATE*, 2019, pp. 928–931.
[4] J. Fabrício Filho, I. B. Felzmann, R. Azevedo, and L. F. Wanner, "AxRAM: A lightweight implicit interface for approximate data access," *FGCS*, vol. 113, pp. 556–570, 2020.
[5] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava, "VarEMU: An Emulation Testbed for Variability-aware Software," in *CODES+ISSS*, 2013.
[6] M. Wyse, A. Baixo, T. Moreau, B. Zorn, J. Bornholt, A. Sampson, L. Ceze, and M. Oskin, "REACT: A Framework for Rapid Exploration of Approximate Computing Techniques," in *Workshop on Approximate Computing Across the Stack*, 2015.
[7] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *MICRO*, 2016, pp. 1–14.
[8] I. B. Felzmann, M. M. Susin, L. Duenha, R. Azevedo, and L. F. Wanner, "ADeLe: A description language for approximate hardware," *FGCS*, vol. 102, pp. 245–258, 2020.
[9] I. Felzmann, J. Fabrício Filho, and L. Wanner, "Risk-5: Controlled approximations for RISC-V," *TCAD*, vol. 39, no. 11, 2020.
[10] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE CAL*, vol. 15, no. 1, 2016.
[11] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, "DRAMPower: Open-source DRAM Power & Energy Estimation Tool."
[12] K. K. Chang, O. Mutlu, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, and H. Hassan, "Understanding Reduced-Voltage Operation in Modern DRAM Devices," *POMACS*, vol. 1, no. 1, 2017.
[13] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *TVLSI*, vol. 27, no. 11, pp. 2629–2640, nov 2019.