

AxNUMA: Approximate Computing in the Linux Kernel via NUMA

Jorge Henrique dos Santos¹, Isaías Felzmann¹

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

j227124@dac.unicamp.br, ibf@unicamp.br

Abstract. *Approximate Computing (AxC) reduces energy and performance overhead for error-tolerant applications. It achieves these efficiency gains by relaxing accuracy requirements. However, its adoption remains constrained by the lack of standard OS support that provides a reliable execution environment. We introduce AxNUMA, a Linux kernel mechanism that abstracts approximate memory into CPU-less NUMA nodes. AxNUMA enables management via standard Linux interfaces without risking kernel stability. Our evaluation using PyTorch ExecuTorch with TinyBERT demonstrates that offloading model weights to approximate NUMA nodes yields reliable system operation and robust inference performance.*

1. Introduction

Approximate Computing is a computing paradigm that proposes trading off the accuracy of results in exchange for benefits such as reduced energy consumption and increased application performance [Dalloo et al. 2024]. This paradigm is based on the premise that certain classes of applications, particularly those that are error-tolerant, can operate even with a controlled level of imprecision without significantly compromising their functionality. Although several studies have explored this concept at different layers of computing, including software, hardware, and architecture [Dalloo et al. 2024], limitations still exist in the support for these techniques at the operating system level [Stazi et al. 2019b, Raha et al. 2016, Fabrício Filho et al. 2020, Dalloo et al. 2024].

A major limitation of using approximate computing techniques in operating systems stems from the very nature of these systems, whose essential function is to manage and control computational resources. Operating systems are not error tolerant [Stazi et al. 2019b], as even low levels of imprecision can result in the corruption of critical kernel data, leading to undefined behaviors. Examples of these behaviors include failures in essential mechanisms, such as synchronization (e.g., locks).

Despite the inherent limitations of operating systems (OSs) regarding approximate computing, they can manage applications that utilize these techniques. OSs must integrate specific mechanisms that allow approximate computing techniques to be exposed and managed safely and efficiently while isolating the kernel from crashes or failures triggered by data errors in approximate memory.

This work was supported by the National Council for Scientific and Technological Development (CNPq); by the Coordination for the Improvement of Higher Education Personnel (CAPES) - Finance Code 001; and by FAEPEX-UNICAMP grants #3770/25 and #2622/25.

An important strategy for incorporating approximate computing techniques into operating systems is the use of approaches based on approximate memory [Raha et al. 2016]. These approaches can be implemented, for example, by controlling the memory refresh rate or through dynamic voltage scaling [Dalloo et al. 2024]. However, the primary challenge associated with this strategy lies in the correct identification and differentiation between critical data (which do not tolerate errors) and error-tolerant data. Moreover, it is essential to ensure that these distinct categories are allocated to separate memory regions, thereby assuring the integrity and proper performance of the operating system [Fabrício Filho et al. 2020]. Previous attempts of incorporating such control capabilities, however, are limited on their flexibility or integration at an operating system level [Stazi et al. 2019a, Raha et al. 2016, Liu et al. 2011, Fabrício Filho et al. 2020].

On the other hand, dealing with heterogeneous memory architectures is not a new concept, as the Non-Uniform Memory Access (NUMA) model is one of the primary solutions for memory tiering [LWN.net 2024a]. NUMA is a memory architecture model widely employed in multiprocessor systems [Jamil et al. 2023], characterized by the physical distribution of memory into different nodes, each associated with a specific set of CPUs. Exploiting NUMA can increase flexibility in the application of approximate memory techniques in the Linux kernel, since nodes may exist that are composed exclusively of memory, without direct association with processors. This model allows for greater control and granularity in data allocation, facilitating the efficient implementation of strategies that utilize approximate memory.

The Linux kernel already offers native support for the NUMA model [Lameter 2013], allowing the system architecture to be described and manual allocations to be performed on specific nodes. Currently, the kernel manages the NUMA architecture based on a single non-uniform characteristic: memory access latency. This parameter depends on the physical proximity between the memory node and the CPU performing the access, as local memory access incurs lower latency than remote memory access.

This paper proposes AxNUMA, whose most fundamental contribution is allowing the Linux kernel to execute safely in systems equipped with approximate memory. To preserve system reliability, AxNUMA adapts the NUMA framework and memory management subsystem to address the specific requirements of approximate computing. It leverages the existing NUMA infrastructure to manage a new non-uniform dimension. Specifically, we propose introducing a new non-uniform characteristic to the memory architecture related to the error level of the memory. This approach is analogous to the distance metric already used to differentiate latencies between nodes.

To validate the system resilience and stability, we integrated our mechanism into the Linux kernel and evaluated its usability using the PyTorch’s ExecuTorch Deep Learning solution [ExecuTorch Contributors 2024]. We demonstrate that by treating approximate memory as a NUMA node, we can safely and transparently offload error-tolerant data structures, such as DNN weights, to approximate regions without compromising overall kernel integrity or triggering system instability. We validated the system robustness by running LLM inference with the TinyBert model [Jiao et al. 2020], proving that approximate memory can be utilized reliably under active deep learning workloads without disrupting the underlying operating system.

This work brings the following contributions:

- A design for safely exposing approximate memory on operating systems without compromising core kernel stability or core OS features.
- A demonstration of standard interface usage to manage approximate resources, eliminating the need for custom allocators.
- A case study evaluating production-grade deep learning frameworks, demonstrating that the proposed abstraction provides the necessary control for error-tolerant computing without sacrificing system reliability or robustness.

2. Background

Approximate computing has the potential to reduce the energy consumption footprint of many error-tolerant applications, including those in machine learning. In the following sections, we discuss its general properties and the main techniques used with a special focus on Approximate Memory techniques. Moreover, we introduce the NUMA architecture and its support in the Linux kernel, highlighting its capability to manage heterogeneous memory systems, which we leverage in this work to enable approximate computing.

2.1. An Overview of Approximate Computing applied to Memories

The core idea of Approximate Computing (AxC) is to achieve gains in energy efficiency and computational performance by introducing a controlled reduction in the precision of computational operation results. This strategy is viable due to the inherent error tolerance of certain applications, such as signal processing and machine learning.

Regarding Approximate Computing techniques, we highlight the comprehensive review conducted by Dalloo et al. [Dalloo et al. 2024]. The authors analyzed studies that applied techniques from the four groups: Data, Software, Architecture, and Circuit. The review indicates that such approaches can significantly contribute to improved energy efficiency across different application types, such as artificial intelligence, IoT, and multimedia processing, where a moderate level of error is often acceptable.

Of particular interest to this work are approximate memory techniques, which present significant potential for enhancing the energy efficiency of computational systems. This is due to the intense communication between processors and memory chips, which, coupled with the immense data volume processed by modern applications, accounts for a major portion of both energy consumption and execution time [Dalloo et al. 2024]. Chen et al. [Chen et al. 2016] observe that systems generally adopt a worst-case design concept, which leads to the over-provisioning of certain parameters (e.g. refresh rate), resulting in higher power consumption. In this context, some studies have investigated relaxing these parameters as a means of reducing power consumption.

In the context of managing approximate memory via the operating system, several works have addressed this problem from both a conceptual and a practical standpoint [Stazi et al. 2019a, Raha et al. 2016, Liu et al. 2011, Fabrício Filho et al. 2020]. Although they present promising results, some limitations have been identified, especially regarding the flexibility of use of the proposed solutions. For example, the work by Stazi et al. [Stazi et al. 2019a] defines specific memory zones with static levels of approximation, which requires management through a custom, non-standard library.

```

1 distance-map {
2     compatible = "numa-distance-map-v1";
3     distance-matrix = <0 0 10>,
4                       <0 1 20>,
5                       <1 1 10>;
6                       <1 0 20>;
7 };

```

Figura 1. NUMA Distance Map. The distance matrix represents the memory access latency between each pair of NUMA nodes.

As presented throughout this section, several research topics within the approximate computing field have reached advanced stages. This work proposes to integrate these research areas by introducing a novel operating system-level approach to safely execute applications, such as production-grade deep learning frameworks, within a unified system architecture.

2.2. Non-Uniform Memory Access (NUMA)

The Non-Uniform Memory Access (NUMA) architecture has become a standard in high-performance servers [Calciu et al. 2017]. In this architecture, the system is composed of nodes, with each node comprising processors, dedicated local memory, and other components. These nodes are interconnected, allowing a processor to access memory belonging to other nodes. However, accessing this remote memory exhibits higher latency compared to accessing local memory, potentially resulting in performance penalties [Lameter 2013].

NUMA (Non-Uniform Memory Access) support was incorporated into the Linux kernel over two decades ago [Lameter 2013]. This integration provides a set of system calls that enable the effective management and optimization of memory utilization in systems based on this architecture [Linux man-pages project 2024]. The kernel also provides mechanisms for memory topology discovery, enabling the identification of which CPUs belong to each NUMA node, which physical memory ranges are associated with each node, and how I/O devices are connected. This topological information can be supplied to the operating system via the ACPI [UEFI Forum 2021] or Device Tree [Linux Kernel Organization] interfaces.

One of the parameters present in the topology description is the distance between nodes. This information is provided via a matrix: the distance map. This map indicates the relative distances between nodes and represents the memory access latency between NUMA nodes to be considered by the Linux kernel to optimize memory allocation. As shown in Figure 1, which represents the NUMA distance map of the system composed by two nodes.

In this work, we propose including a new matrix, the approximation map, to the topology. The approximation map represents the error level associated with each node. The goal is for this new matrix to be interpreted by the kernel during the memory allocation process, enabling error-aware decisions regarding the precision level of the memory regions utilized.

3. Related Work

Approximate Memory in the Linux Kernel. Regarding support for Approximate Computing (AxC) within the Linux kernel, we highlight the work by Stazi et al.

[Stazi et al. 2019b]. Their paper presents a concrete implementation of this approach in the Linux kernel by introducing modifications to memory management to accommodate different error levels. The proposed solution adds up to four approximate memory zones, each associated with a specific level of error, to the standard Linux memory zones. In this manner, the authors extend the kernel’s memory configuration to include regions explicitly designated for approximate computation.

As for the user interface, the work proposes the `aproxmalloc()` function. This function allows user-space applications to request data allocation within specific approximate memory zones, offering a more fine-grained control over the utilization of these resources. However, their proposed solution exhibits limitations regarding flexibility. This limitation lies in the static configuration of the error level for each approximate memory zone. While the zones are hierarchically organized according to their error level, there is no support for dynamic adjustment of these levels. This restriction limits the solution’s adaptability to different scenarios and application requirements. Unlike a static approach with a fixed number of approximate memory zones, AxNUMA scales to support an arbitrary number of approximation levels. By leveraging the native Linux NUMA API rather than custom allocation libraries, our solution integrates seamlessly with standard system management tools.

Approximate Memory in Operating Systems. Although they do not directly explore the integration of approximate memory into the Linux kernel, the works by Liu et al. [Liu et al. 2011] and Fabrício Filho et al. [Fabrício Filho et al. 2020] conceptually address the problem of adding support for approximate memory in operating systems. The authors propose solutions to reduce the power consumption of DRAM and SRAM memories by means of refresh rate control [Liu et al. 2011] or operating voltage adjustment [Fabrício Filho et al. 2020]. As a shared aspect, the authors suggest distinguishing between critical and non-critical data, allocating them to memory regions with varying error susceptibilities. To enable this approach within the operating system, the authors conceptualize the incorporation of the error level for each memory region into the structure of each page table entry. This information would then indicate to the operating system in which memory bank region each page should be allocated.

Although these proposals offer relevant conceptual contributions, the studies do not present an implementation or validation of this idea at the operating system level. The evaluation was conducted by executing each application in isolation, meaning without a complete operational environment. Despite this limitation, the results indicate a reduction of 20–25% in DRAM or SRAM power consumption, thus demonstrating the significant potential of the approach.

Finally, the work by Raha et al. [Raha et al. 2016] also presents a relevant contribution to the field of approximate memory. The core characteristic of their proposal consists of utilizing a single refresh rate for the entire memory. The authors emphasize that this approach simplifies the practical system implementation by reducing the complexity associated with managing multiple memory banks with distinct refresh rates. The study was implemented on a lightweight operating system (uC/OS-II) running on an Altera Stratix IV FPGA with 1GB of DDR3 DRAM. The authors observed a 70% reduction in power consumption during refresh operations.

Despite these contributions, this approach does not guarantee system stability during prolonged operations, since it is based on a single memory with a fixed refresh rate. Even when allocating critical data to lower error-rate regions, random errors can gradually compromise the kernel’s integrity, potentially leading to catastrophic failures. Furthermore, although this strategy simplifies memory management, it restricts more refined optimizations, as different applications may require distinct levels of error tolerance. This limitation makes the system less adaptable to heterogeneous scenarios, where flexibility in resource allocation is crucial for maximizing energy efficiency gains.

NUMA and Unconventional Memories. Regarding the use of NUMA to handle unconventional memories, this is an approach that has been applied to manage systems composed of memories with varying access and performance characteristics [LWN.net 2024b]. For example, this includes support for persistent memories(PMEM), where each persistent memory device is represented as a CPU-less NUMA node [Balcer and Andrejczuk , LWN.net 2019]. Other works have addressed NUMA support to manage memory based on characteristics beyond just distance, incorporating parameters such as latency, bandwidth, and power consumption [Williams et al. 2017, Khandual 2017].

4. AxNUMA: Representing Approximation in NUMA Nodes

Our primary design goal with AxNUMA is to safely bridge the usability gap in approximate computing by elevating approximate memory from a specialized hardware peripheral to a first-class operating system resource without compromising kernel functionality or system stability. Rather than creating new, non-standard system calls (e.g., a `malloc_approx`), we leverage the inherent flexibility of the Linux NUMA subsystem to manage approximate resources safely alongside standard memory. This section details the architectural abstraction of approximate memory as CPU-less NUMA nodes, the mechanism for isolating kernel data from approximation errors and the policy-based interface exposed to user-space applications.

4.1. Proposed NUMA Abstraction

In traditional multi-socket systems, NUMA nodes are described by their access latency relative to a specific CPU. The Linux kernel uses a “distance” metric to make allocation decisions, prioritizing memory that is local to the execution thread.

We extend this model to represent the level of approximation as a topological property. In addition to the distance map, we introduce the approx-map, a new matrix that specifies the approximation level for each NUMA node. Figure 2 illustrates how a topology consisting of two NUMA nodes can be described. For example, Node 0 operates without approximation, while Node 1 has approximation enabled, identified by the non-zero approximation level.

We architect the system such that approximate memory banks are exposed to the kernel as distinct, CPU-less NUMA nodes (often referred to as “memory-only nodes” in the context of CXL or PMEM). By decoupling the approximate memory from the CPU’s local DRAM node, we achieve two architectural benefits:

- **Logical Isolation:** The approximate memory has a unique Node ID, allowing the OS to address it as a discrete pool of physical pages.

```

1 approx-map {
2     compatible = "numa-approx-map-v1";
3     approx-matrix = <0 0>,
4                   <1 1>;
5 };

```

Figura 2. NUMA approx-map. The approximation matrix represents the approximation level of each NUMA node.

- **Default Avoidance:** By assigning these nodes a non-zero approximate level, the kernel’s default allocation process avoids allocating memory from these nodes. This ensures that standard system processes and non-approximate applications never accidentally allocate unsafe memory.

4.2. Kernel Safety and Fault Isolation

Although NUMA abstractions effectively handle hardware heterogeneity, they are insufficient for ensuring reliable kernel operation over approximate memory. A critical challenge in OS-managed approximate memory is preventing fatal system crashes. If kernel data structures (e.g., page tables, inodes, or process descriptors) are placed in approximate memory, a single bit-flip could trigger kernel malfunction.

In order to ensure kernel safety, we modified the memory allocators to prevent sensitive data structures from being placed on approximate nodes. Consequently, from the early boot phase using the memblock allocator through to the “normal” memory management stage with the buddy and slab allocators, critical kernel data is always allocated on reliable memory.

In summary, memory is allocated on the approximate node only when explicitly requested by the NUMA-related allocation call. Furthermore, even the NUMA node metadata and the page table pages associated with approximate memory, that would usually be stored in the same NUMA node to the data they refer, are always allocated on a reliable node.

4.3. Usability via Standard Interfaces

The core contribution of AxNUMA is the seamless integration of approximate memory management with existing Linux memory policies and APIs. Since the approximate memory is exposed as a standard NUMA node (e.g., Node 1), developers and administrators can use the existing Linux memory libraries, system calls and policy mechanisms without any modifications, enabling easy adoption and compatibility with current tools and workflows. Figure 3 demonstrates an example program where the input and output matrices are explicitly placed in the approximate memory region. By replacing standard `malloc` or `calloc` calls with `numa_alloc_onnode`, the developer can enforce that these data structures are stored on a approximate memory.

5. Implementation

Our implementation targets the Linux Kernel (v6.11.4) and is restricted to the RISC-V architecture, focusing on enabling safe access to approximate memory without destabilizing the system. The modifications span three critical subsystems: firmware description (Device Tree), early boot memory management (memblock) and the runtime page allocator.

```

1 #include <numa.h>
2 #include <numaif.h>
3
4 #define APPROX_NODE 1 // Logical node mapped to approximate memory
5
6 int main() {
7
8     // Allocate matrices on the Approximate Memory Node
9     // Using numa_alloc_onnode() instead of malloc()
10    A = (int *) numa_alloc_onnode(bytes, APPROX_NODE);
11    B = (int *) numa_alloc_onnode(bytes, APPROX_NODE);
12    C = (int *) numa_alloc_onnode(bytes, APPROX_NODE);
13
14    // Initialize memory and execute an algorithm (i.e Matrix
15        Multiplication)
16
17    numa_free(A, bytes);
18    numa_free(B, bytes);
19    numa_free(C, bytes);
20 }

```

Figure 3. Example program using AxNUMA. The data pointers are explicitly allocated in approximate memory using standard NUMA allocation calls.

5.1. Hardware Description via Device Tree

To integrate approximate memory into the OS without introducing non-standard memory node types, we leverage the existing Device Tree (DT) NUMA topology definitions. We utilize standard memory nodes associated with specific NUMA IDs and introduce a new property: the `approx-map`. Modeled directly after the standard distance-map binding used in NUMA definitions, `approx-map` defines an approximation matrix. This matrix associates specific NUMA node IDs with their corresponding approximation levels (e.g., bit error rates or refresh intervals) as shown in figure 2 .

During the boot sequence, the kernel parses this map alongside the standard NUMA distance tables. This approach allows us to "tag" standard memory banks as approximate simply by associating them with a node ID present in the `approx-map`. This topology-based description provides a seamless way to quantify reliability levels across the system without requiring custom physical memory drivers.

5.2. Early Boot Reservation (Memblock)

A significant challenge in managing heterogeneous memory is preventing the kernel from allocating critical data structures on unreliable media before the full memory management subsystem is initialized. The Linux memblock allocator governs this early boot phase.

By default, memblock considers all discovered memory as valid candidates for kernel text, data, and initial page tables. We modified the memblock initialization routine to enforce a strict reservation policy based on the topology defined in Section 4.1. During the early scan of physical memory nodes, our modified kernel cross-references the NUMA node ID of each memory region against the parsed `approx-map`. To guarantee strict isolation, we explicitly map these physical ranges of approximate nodes to `ZONE_MOVABLE`.

By restricting approximate memory to this specific zone, we leverage the kernel’s inherent guarantee that `ZONE_MOVABLE` is never used for pinned kernel data structures (such as page tables, slab caches, or kernel stacks). This effectively “hides” the approximate memory from the kernel’s core operational needs while keeping it available for higher-level user allocations. This ensures that no critical system structures are ever placed on unreliable media during the boot process.

5.3. Runtime Page Allocator (Memory Management)

The most significant change is adapting the Memory Management (MM) subsystem to prioritize standard memory for critical kernel structures. To achieve this, we tweaked architecture-specific page table allocation, allocation policies, and movable zones.

Page Table Allocation: We updated RISC-V’s page table allocation functions (`pgd_alloc`, `p4d_alloc_one`, etc.) to be node-aware, ensuring page table pages stay on error-free nodes and are not stored on approximated nodes as they have control data about the pages themselves.

Allocation Policies: We modified the core page allocation logic to default to `def_alloc_nodemask`, which is composed by nodes with no approximation, if no specific `nodemask` is provided. This prevents general allocations from falling into approximate memory. Moreover, we force the SLUB allocator and memory interleaving policies to respect the `def_alloc_nodemask`.

Movable Zones: Any NUMA node with a non-zero approximation level is automatically treated as “Movable” memory. This ensures that only user-space data (which can be moved or reclaimed) is placed there, while the kernel core stays in reliable memory.

6. Experimental Validation

We evaluate AxNUMA using PyTorch’s Executorch, a standard Deep Learning framework, to demonstrate that applications can leverage approximate memory within a Linux environment without compromising overall system integrity. Moreover, we tested the application viability by testing whether we can offload LLM’s model weights to approximate regions while showing a potential balance between system efficiency and accuracy degradation.

6.1. Experimental Setup

Since commercial approximate memory hardware is not yet widely available, we emulate the architectural behavior of approximate DRAM using a virtualized environment. We utilize the AxPIKE simulator [Felzmann et al. 2021] integrated with the AxRAM interface [Fabrício Filho et al. 2020]. While AxPike enables the exploration of approximate computing techniques, the AxRAM interface facilitates controlled error injection during DRAM operations. Our evaluation is conducted on a simulated RISC-V processor using this integrated AxPike–AxRAM solution. Moreover, the NUMA topology is composed by a CPU-less node that houses the approximate memory region, in addition of the standard one as presented by Figure 4.

System Configuration:

- **Host Hardware:** 12th Gen Intel(R) Core(TM) i7-12800H (2.40 GHz), 32GB DDR4.

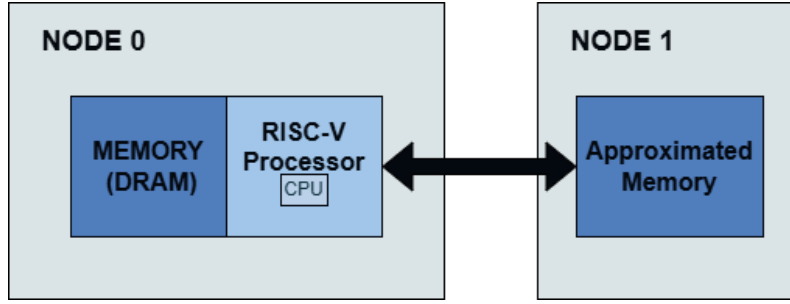


Figura 4. Experimental Topology. NUMA system architecture comprising a standard node and a CPU-less node containing approximate memory.

- **Guest OS:** Linux Kernel v6.11.4 (modified with our NUMA emulation patches).
- **Topology**
 - Node 0 (Reliable Tier): 1 vCPUs, 8GB DRAM. This node hosts the kernel image, page tables, and latency-sensitive application code.
 - Node 1 (Approximate Tier): 0 vCPUs, 8GB DRAM. This is a "memory-only" node configured with high Approximate level values to prevent automatic allocation by the standard scheduler.

To quantify the impact of approximation on application output, we adopted a software-based fault injection mechanism similar to the approach described by Gao et al. [Gao et al. 2024]. Their methodology leverages approximate memory to reduce energy consumption in Large Language Models (LLM) by injecting random bit-level errors across the weights of quantized models. In this work, we evaluate the performance of the modified DNNs under a Bit Error Rate (BER) ranging from 10^{-7} to 10^{-2} , consistent with the previous works from Gao et al. [Gao et al. 2024] and Nguyen et al. [Nguyen et al. 2018].

6.2. Workload

We selected a production-grade deep learning framework to validate the usability and viability of our abstraction: PyTorch’s ExecuTorch [ExecuTorch Contributors 2024]. ExecuTorch is a lightweight, modular middleware for deploying PyTorch models to edge devices. Its minimal C++ runtime and backend delegation provide efficient hardware acceleration across NPUs, DSPs and microcontrollers with low memory overhead.

To demonstrate the framework’s efficacy in a real-world edge computing scenario, we employed the TinyBERT architecture, specifically the gokuls/BERT-tiny-emotion-intent [Gokulakrishnan 2022] variant. This model was fine-tuned on the dair-ai/emotion dataset [Saravia et al. 2018], providing a robust baseline for multi-class emotion classification within a highly constrained parameter space. By deploying this compressed Transformer model via ExecuTorch, we evaluate the system’s ability to maintain inference accuracy while adhering to the strict latency and power benchmarks characteristic of embedded environments. This setup follows the methodology established by Gao et al. [Gao et al. 2024] to provide a robust baseline for comparison.

To validate our solution for approximate memory, we extended the core memory management logic of the ExecuTorch runtime. Specifically, we modified the primary CPU

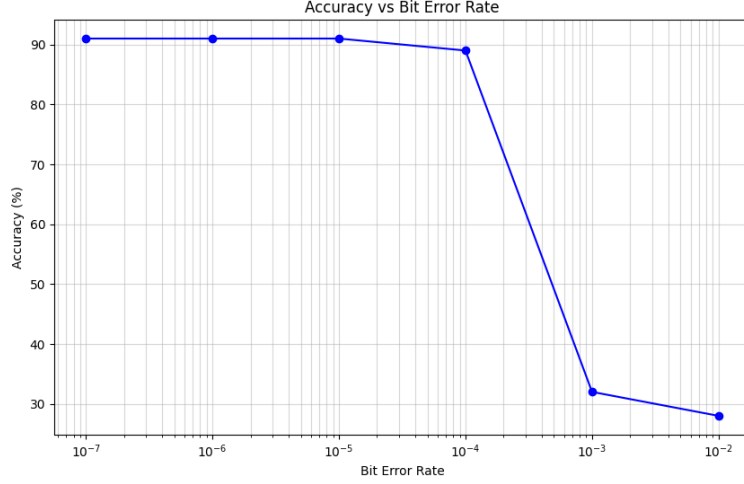


Figure 5. TinyBERT’s Accuracy as function of BER. Impact of approximate memory-induced bit errors on Large Language Model (LLM) inference accuracy.

acceleration engine, XNNPACK, to store the model weights on the specific approximate-memory NUMA node.

6.3. Metrics

The system is evaluated by assessing Application Fidelity, defined as the end-to-end impact of approximation on model performance. For the dair-ai/emotion dataset, we measure the degradation in classification accuracy across varying Bit Error Rates (BER). This metric quantifies the robustness of the TinyBERT model when deployed on approximate NUMA nodes under different noise conditions.

6.4. Results

To evaluate the application fidelity of the proposed AxNUMA abstraction, we measured the end-to-end classification accuracy of the TinyBERT model across a Bit Error Rate (BER) ranging from 10^{-7} to 10^{-2} , as illustrated in Figure 5.

The experimental results demonstrate that the model exhibits substantial fault tolerance at lower error injection levels. Specifically, at BERs of 10^{-7} and 10^{-5} , the system maintains a robust baseline accuracy of approximately 91%. This high resilience persists up to a BER of 10^{-4} , where the model experiences only a marginal degradation, sustaining an accuracy of nearly 89%. This indicates that the AxNUMA mechanism can successfully expose approximate memory resources for inference tasks without significantly compromising application output at these error levels.

In contrast, the study by Gao et al. [Gao et al. 2024] suggests that TinyBERT can maintain near-original accuracy even at a BER as high as 10^{-2} . The primary factor accounting for this divergence could be the difference between true fixed-point execution and floating-point emulation. The reference study evaluates resilience by ”recovering”INT8 weights back to a Float16 format for inference. In the other hand, our imple-

```

1 static void* xnn_numa_allocate(void* context, size_t size) {
2     return numa_alloc_onnode(size, APROX_NODE);
3 }
4
5 const struct xnn_allocator xnn_numa_allocator = {
6     .allocate = xnn_numa_allocate,
7     .deallocate = xnn_numa_deallocate
8 };

```

Figura 6. AxNUMA usage. Executorch’s memory operations interface to use AxNUMA.

mentation utilizes the XNNPACK backend, which executes true integer-based memory operations and arithmetic.

The AxNUMA mechanism facilitates significant energy reductions at the hardware level. Considering a power-scaling characterization of 28nm CMOS SRAM Reviriego et al. [Reviriego et al. 2021], a target Bit Error Rate of 10^{-4} enables 50.0% dynamic power savings and 30.0% static power savings.

These findings highlight the viability of the AxNUMA abstraction in managing approximate memory resources within a Linux environment without compromising system stability, as the platform reliably sustained continuous LLM workloads even under elevated error rates.

From a usability perspective, AxNUMA significantly reduces implementation complexity by leveraging the existing Linux NUMA infrastructure. Integrating approximation support requires only a straightforward transition from standard memory allocation functions (i.e `malloc`), to NUMA system calls (i.e `numa_alloc_onnode`). As illustrated in Figure 6, in the context of the presented experiment, this was achieved by implementing a NUMA version of a memory operations interface used by Executorch allowing applications to target approximate memory nodes with minimal code modification.

7. Conclusion

This paper introduced AxNUMA, a novel operating system-level framework for managing approximate memory within the Linux kernel without compromising core OS functionality. By leveraging existing NUMA support to expose approximate memory as CPU-less nodes, AxNUMA enables seamless integration with standard Linux interfaces and memory policies.

Unlike previous approaches, AxNUMA eliminates the need for specialized libraries, significantly improving developer usability. Our evaluation using Deep Learning workloads demonstrates that AxNUMA effectively supports real-world applications, providing a consistent and robust method to control the usage of approximate memories in general-purpose, Linux-powered architectures. The modified Linux kernel source code is publicly available in the GitHub repository at <https://github.com/JorgeSantos18/linux/tree/numa-aprox>.

Referências

- Balcer, P. and Andrejczuk, G. Memory tiering (part 1).
- Calciu, I., Sen, S., Balakrishnan, M., and Aguilera, M. K. (2017). Black-box concurrent data structures for numa architectures. *ACM SIGPLAN Notices*, 52(4):207–221.
- Chen, Y., Yang, X., Qiao, F., Han, J., Wei, Q., and Yang, H. (2016). A multi-accuracy-level approximate memory architecture based on data significance analysis. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 385–390. IEEE.
- Daloo, A. M., Humaidi, A. J., Al Mhdawi, A. K., and Al-Raweshidy, H. (2024). Approximate computing: Concepts, architectures, challenges, applications, and future directions. *IEEE Access*.
- ExecuTorch Contributors (2024). Executorch: Pytorch’s on-device ai stack. <https://github.com/pytorch/executorch>.
- Fabício Filho, J., Felzmann, I. B., Azevedo, R., and Wanner, L. F. (2020). Axram: A lightweight implicit interface for approximate data access. *Future Generation Computer Systems*, 113:556–570.
- Felzmann, I., Fabício Filho, J., and Wanner, L. (2021). Axpique: instruction-level injection and evaluation of approximate computing. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 491–494. IEEE.
- Gao, Z., Deng, J., Reviriego, P., Liu, S., and Lombardi, F. (2024). Reducing the energy dissipation of large language models (llms) with approximate memories. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.
- Gokulakrishnan, S. (2022). Bert-tiny-emotion-intent. <https://huggingface.co/gokuls/BERT-tiny-emotion-intent>.
- Jamil, H., Chung, J., Bicer, T., Kosar, T., and Kettimuthu, R. (2023). Throughput optimization with a numa-aware runtime system for efficient scientific data streaming. In *Proceedings of the SC’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 795–805.
- Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. (2020). Tinybert: Distilling bert for natural language understanding. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 4163–4174.
- Khandual, A. (2017). Hierarchical numa design.
- Lameter, C. (2013). Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51.
- Linux Kernel Organization. Numa binding description. <https://www.kernel.org/doc/Documentation/devicetree/bindings/numa.txt>. Accessed: 2025-03-21.
- Linux man-pages project (2024). numa(7) — linux manual page. <https://man7.org/linux/man-pages/man7/numa.7.html>. Accessed: 2025-03-21.
- Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B. G. (2011). Flickr: Saving dram refresh-power through critical data partitioning. In *Proceedings of the sixteenth inter-*

- national conference on Architectural support for programming languages and operating systems*, pages 213–224.
- LWN.net (2019). Persistent memory as ram. <https://lwn.net/Articles/787418/>. Accessed: 2025-03-22.
- LWN.net (2024a). Better support for locally-attached-memory tiering. <https://lwn.net/Articles/974126/>. Accessed: 2025-03-18.
- LWN.net (2024b). Extending the mempolicy interface for heterogeneous systems. <https://lwn.net/Articles/973964/>. Accessed: 2025-03-22.
- Nguyen, D. T., Kim, H., Lee, H.-J., and Chang, I.-J. (2018). An approximate memory architecture for a reduction of refresh power consumption in deep learning applications. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.
- Raha, A., Sutar, S., Jayakumar, H., and Raghunathan, V. (2016). Quality configurable approximate dram. *IEEE Transactions on Computers*, 66(7):1172–1187.
- Reviriego, P., Liu, S., Ertl, O., Niknia, F., and Lombardi, F. (2021). Computing the similarity estimate using approximate memory. *IEEE Transactions on Emerging Topics in Computing*, 10(3):1593–1604.
- Saravia, E., Liu, H.-C. T., Huang, Y.-H., Wu, J., and Chen, Y.-S. (2018). CARER: Contextualized affect representations for emotion recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3687–3697, Brussels, Belgium. Association for Computational Linguistics.
- Stazi, G., Mastrandrea, A., Olivieri, M., and Menichelli, F. (2019a). Full system emulation of approximate memory platforms with appropinquo. *Journal of Low Power Electronics*, 15(1):30–39.
- Stazi, G., Mastrandrea, A., Olivieri, M., and Menichelli, F. (2019b). Quality aware approximate memory in risc-v linux kernel. In *2019 15th Conference on Ph. D Research in Microelectronics and Electronics (PRIME)*, pages 177–180. IEEE.
- UEFI Forum (2021). Acpi specification - section 17: Numa architecture platforms. Technical report, Unified Extensible Firmware Interface (UEFI) Forum. Accessed: 2025-03-21.
- Williams, S., Ionkov, L., and Lang, M. (2017). Numa distance for heterogeneous memory. In *Proceedings of the Workshop on Memory Centric Programming for HPC, MCHPC’17*, page 30–34, New York, NY, USA. Association for Computing Machinery.