CheckMate: LLM-Powered Approximate Intermittent Computing

Abdur-Rahman Ibrahim Sayyid-Ali
25100204@lums.edu.pk
LUMS
Pakistan

Abdul Rafay
abdul.rafay@lums.edu.pk
LUMS
Pakistan

Muhammad Abdullah Soomro**
24100180@lums.edu.pk
LUMS
Pakistan

Muhammad Hamad Alizai
hamad.alizai@lums.edu.pk
LUMS
Pakistan

Naveed Anwar Bhatti
naveed.bhatti@lums.edu.pk
LUMS
Pakistan

Abstract

Batteryless IoT systems face energy constraints exacerbated by checkpointing overhead. Approximate computing offers solutions but demands manual expertise, limiting scalability. This paper presents CheckMate, an automated framework leveraging LLMs for context-aware code approximations. CheckMate integrates validation of LLM-generated approximations to ensure correct execution and employs Bayesian optimization to fine-tune approximation parameters autonomously, eliminating the need for developer input. Tested across six IoT applications, it reduces power cycles by up to 60% with an accuracy loss of just 8%, outperforming semi-automated tools like ACCEPT in speedup and accuracy. CheckMate's results establish it as a robust, user-friendly tool and a foundational step toward automated approximation frameworks for intermittent computing.

CCS Concepts

• **Computer systems organization → Embedded software**; • **Computing methodologies → Information extraction**.

Keywords

Intermittent Computing, Approximate Computing, Large Language Models (LLMs), Batteryless IoT

ACM Reference Format:
Abdur-Rahman Ibrahim Sayyid-Ali, Abdul Rafay, Muhammad Abdullah Soomro*, Muhammad Hamad Alizai, and Naveed Anwar Bhatti. 2025. CheckMate: LLM-Powered Approximate Intermittent Computing. In *The 23rd ACM Conference on Embedded Networked Sensor Systems (SenSys '25), May 6–9, 2025, Irvine, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3715014.3722056

**Now pursuing a PhD at UMass Amherst.

SenSys '25, May 6–9, 2025, Irvine, CA, USA

ACM ISBN 979-8-4007-1479-5/25/05
https://doi.org/10.1145/3715014.3722056

1 Introduction

Energy harvesting enables batteryless IoT devices to operate without regular batteries, reducing maintenance costs and supporting multi-year unattended deployments [1]. However, environmental energy sources are often erratic, leading to frequent and unpredictable power failures, emphasizing the critical importance of energy efficiency in batteryless IoT applications [3, 4, 33, 39].

Traditionally, checkpointing methods have served as the cornerstone for managing program state across power failures in energy-harvesting devices, despite introducing considerable overhead [5, 13, 14, 31, 52]. This computing paradigm, which frequently saves and restores program state, is commonly referred to as *intermittent computing*. Over the past decade, researchers have sought to mitigate this overhead through various innovations, such as optimizing checkpoint placement [2, 14, 41], virtualizing memory [42], fragmenting tasks into idempotent code blocks [17, 40], and employing event-driven programming models [64]. Despite these efforts, the marginal gains from further optimization are diminishing, signaling that the traditional focus on reducing checkpointing overhead in intermittent computing has reached its practical limits [6].

In this work, we shift the paradigm from minimizing checkpoint overhead to directly reducing the energy consumed by computations. Leveraging approximate computing techniques, we relax computational accuracy in targeted ways, which lowers energy demands and enables more tasks to be completed per power cycle[1], and reduces the need for frequent checkpoints. This strategy is particularly advantageous for intermittent computing on batteryless IoT systems, where energy availability is unpredictable and resources are limited. In such environments, slight compromises in accuracy are acceptable to achieve task completion with reasonable precision [10, 37], making approximate computing a suitable solution for applications such as filtering, machine learning, image processing, and data fusion. These areas are essential to many IoT systems, where efficient computation is critical to maintaining functionality under extreme energy constraints.

Challenges: While several researchers have already introduced approximation techniques in batteryless systems, signaling the early sparks of this paradigm shift, the path to broader adoption

[1]A power cycle refers to a single cycle of intermittent computing during which the system is powered *ON* and performs computations

remains obstructed. The absence of an automated framework for efficiently implementing these approximations stands as the primary challenge. Researchers have been limited to optimizing approximation techniques for specific applications and scenarios, resulting in studies that are replicable only within those exact contexts [10, 11, 22, 29, 30, 32, 37, 55, 63]. At its core, the challenge revolves around transforming the abstract potential of approximation into practical solutions that work across diverse applications. Developers face a fundamental dilemma: how to achieve the right balance between energy savings and computational precision without undermining the reliability of program output. We have distilled this challenge into three key areas that require attention to advance the field.

1 Identifying suitable opportunities for approximation requires more than just technical know-how; it demands a nuanced understanding of the application's semantics and execution context, hardware architecture, and the particularities of various approximation strategies. Poor judgment in this process can compromise energy efficiency or, worse, result in inaccuracies that degrade system performance. Developers must navigate this terrain carefully, ensuring that the energy-accuracy trade-offs they introduce are both purposeful and effective.

2 Even when approximation opportunities are identified, the next hurdle lies in the painstaking process of manual tuning. This step is often labor-intensive, requiring developers to carefully tailor adjustments to match the specific needs of each application. This fine-tuning process, while essential, is prone to errors and demands significant time and effort. Developers must walk a tightrope, ensuring that the savings in energy consumption do not come at the expense of unacceptable losses in computational fidelity.

3 The challenges do not end with implementation. Even the most thoughtfully designed approximations must withstand the unpredictable realities of real-world deployment. Predicting how these approximations will behave post-deployment, under varying conditions and across multiple use cases, introduces yet another layer of complexity. Without a way to accurately foresee the impact of approximations, developers risk deploying systems that fail to meet performance expectations, compromising both scalability and reliability.

Contribution: We introduce CheckMate[2], an open-source novel automated framework that harnesses the intelligence of Large Language Models (LLMs) to effectively balance energy consumption and computational accuracy in batteryless IoT systems. Unlike existing frameworks constrained by laborious manual tuning [56] or simplistic one-size-fits-all strategies [36], CheckMate breathes sophistication into the process by merging LLM-driven insights with automated validation and optimization mechanisms. LLMs' semantic understanding and contextual awareness enable our approach to identify effective points for approximation, determine suitable techniques, and specify their application. However, today's LLMs have inherent limitations: they cannot guarantee that the modified code for embedded platforms compiles correctly, assess the energy requirements of the target platform to estimate power cycles, or ensure an efficient trade-off between energy efficiency and computational accuracy. To address these challenges, CheckMate integrates a rigorous validation process to verify the correctness of LLM-generated code. This process detects compile-time and runtime errors, iteratively feeding them back to the LLM for correction until the code functions as intended. Once validated, the code is subjected to fine-tuning through Bayesian optimization, a probabilistic model-based approach, within a cycle-accurate simulator designed for intermittent computing. This optimization identifies the balance point that minimizes power cycles while ensuring computational accuracy remains within user-defined error bounds. This whole process is depicted in Figure 1 and described in Section 2.

Benefits: The effectiveness of CheckMate is validated through a comprehensive evaluation encompassing six diverse IoT applications, five energy harvesting traces, and both simulation and real-world scenarios. CheckMate achieved significant energy efficiency, reducing power cycles by 15–60% while maintaining error rates within the range of 6–25%. Compared to semi-automated frameworks like ACCEPT [56], CheckMate demonstrates comparable or superior performance while eliminating the need for expert intervention. In real-world hardware evaluations, results closely align with simulation predictions, reinforcing the tool's reliability. Additionally, a user study involving computer science students highlights its practicality, with a 5x reduction in time to apply approximations and a significant decrease in error rates compared to manual methods. This remarkable performance was attained through a seamless *one-click* process, which obviates the need for the labor-intensive fine-tuning typically required by existing frameworks.

As developers increasingly seek ways to optimize batteryless applications, CheckMate offers a glimpse into the future; where intelligent automation, backed by LLMs, transforms the approach to approximate intermitted computing for batteryless IoT.

2 Design Choices and Workflow

The development of CheckMate introduced several challenges that prompted iterative design decisions. In the following, we outline the key design choices and lessons learned through micro-experiments and continuous refinements.

2.1 Error-resolution feedback loop

In the initial stages of CheckMate's development, we began with a simple experiment: entrusting the LLM, with its purported magic-like powers, to approximate the code for execution on a batteryless platform. The objective was for the LLM to process the entire codebase in one submission and return the approximated version. While the LLM demonstrated a rudimentary ability for code approximation, it frequently produced code riddled with compilation or runtime errors. The generated code required manual adjustments for seamless integration, which contradicted our objective to minimize developer intervention. This outcome highlighted the need for a robust error-resolution mechanism within CheckMate.

Approach: We implemented an iterative feedback mechanism for CheckMate to effectively mitigate errors introduced by LLM-generated code. Each iteration involved interaction between the

[2]The name 'CheckMate' embodies the tool's core functionality, automatically integrating *CHECK*pointing and approxi*MATE* computing techniques. Source code is publicly available at https://github.com/SYSNET-LUMS/CheckMate

LLM and CheckMate's external script, which compiled and executed the LLM-generated approximated code. Compilation failures and runtime errors were systematically logged and fed back to the LLM, creating a cycle of autonomous refinements and correction. This iterative process marks a foundational step towards achieving a one-click solution, as it eliminates the need for manual debugging.

2.2 Dynamic tuning via adjustable knobs

A key challenge was enabling the LLM to understand the impact of its approximations on both output accuracy and energy efficiency. Today's LLMs cannot measure hardware energy consumption or assess the resulting trade-offs between output accuracy and energy usage. This limitation necessitated the integration of an external simulator to provide the essential feedback needed for informed decision-making.

Building on the initial error-resolution feedback loop, we introduced a secondary feedback mechanism leveraging a cycle-accurate simulator. The application is executed inside this simulated environment using user-provided input traces. The simulator evaluates output error rates and power cycle reductions, producing quantitative metrics that the LLM itself cannot compute. This approach bridges the gap between the LLM's capabilities and the requirements for optimizing approximations.

However, the fine-tuning process did not proceed as anticipated, frequently oscillating between overly conservative and overly aggressive adjustments. This behavior complicates the achievement of an effective balance and prolongs feedback cycles, increasing both the cost of LLM interactions and the computational workload of running the simulator. We needed a fine-grained exploration of the LLM-suggested approximation techniques and their impact on errors and energy consumption, a task that LLMs were ill-suited for due to their inefficiency in handling such domain-specific optimization challenges.

Approach: We significantly revised our approach to the fine-tuning process by redefining the role of the LLM. Instead of relying on it to produce hardcoded approximation parameters, we prompted it to introduce adjustable *knobs* within the code to control approximation levels, offloading unnecessary burdens from the LLM.

For each approximation opportunity, such as the truncation factor in loop perforation, the LLM was guided to embed these knobs (variables) into the codebase along with their suggested ranges. This design enables precise control over the degree of approximation for specific functions, allowing targeted tuning to adhere to defined error tolerances.

To automate the process of identifying the optimal knob values, we employed Bayesian optimization, which determines knob configurations that achieve the desired balance between energy efficiency and computational accuracy, accounting for the interactions among multiple knobs. The knobs also serve as mechanisms to mitigate inaccurate approximations by enabling the optimization algorithm to 'tune out'[3] the approximation.

[3]The optimization algorithm can adjust the knob to a value that ensures the corresponding code section behaves as it would in the original, unmodified version, effectively nullifying the approximation.

2.3 Context-aware Chain-of-Thought

Another challenge was the tendency of LLMs to miss critical approximation opportunities or apply suboptimal approximations, often leading to an unnecessary increase in the number of knobs leading to a substantial increase in the runtime of Bayesian optimization without proportional benefits. This issue primarily stemmed from the LLM's limited ability to holistically analyze the code structure when processing an entire codebase in one step, restricted by inference time and token generation limits. Furthermore, analyzing the entire codebase disrupted the LLM's tracking of the execution sequence of different functions in the code, which is essential for applying effective approximations. This context loss frequently led to *hallucinations*, where the LLM generated irrelevant or incorrect outputs.

Approach: To address these issues, multiple strategies are available, including *chain-of-thought* (CoT) prompting with few-shot learning, fine-tuning models with additional data, or implementing retrieval-augmented generation (RAG) to contextualize LLM outputs. Due to the absence of sufficiently large and relevant datasets on approximate computing, running a specialized fine-tuned model or one equipped with retrieval-based generation is impractical. Instead, we adopt a context-aware CoT strategy with few-shot learning to enhance the quality of the LLM's code output. This approach allows us to supplement the LLM's inherent reasoning with explicit problem-solving steps that align with expert decision-making, guiding the model toward more accurate and contextually appropriate outputs. Additionally, incorporating a few carefully selected examples of approximations helps the LLM recognize how knobs and approximation techniques should be structured within the code

The strategy begins with *purpose identification*, where the LLM determines the application's primary objective and provides summaries for each function. This step establishes a high-level understanding of the codebase context, which is logged for later use. The LLM then generates a list of functions suitable for approximation, serving as a filter to avoid suboptimal modifications, ensuring that the LLM performs context-aware function selection, providing a foundation for targeted approximations. As the process continues, the conversation history may exceed the LLM's context window, leading to a loss of initial context. To mitigate this, subsequent interactions occur in a new conversation, where the LLM is reintroduced to the codebase using only the previously generated summaries and the list of functions selected for approximation.

We iterate over the selected functions using a function call graph (FCG) to guide the process in a sequential order. For each selected function, the LLM performs two tasks. First, it identifies potential approximation strategies and reasons for their impact on the codebase. Next, based on this reasoning, the LLM is tasked with implementing the most suitable approximation for the selected functions. This context-aware strategy enables precise and effective approximations tailored to the application's operational requirements.

2.4 Workflow

Each design choice in developing CheckMate stemmed from addressing practical challenges and was iteratively refined through experimentation. The integration of *feedback loops*, the deployment of *knobs*, the use of *Bayesian optimization*, and the adoption of a

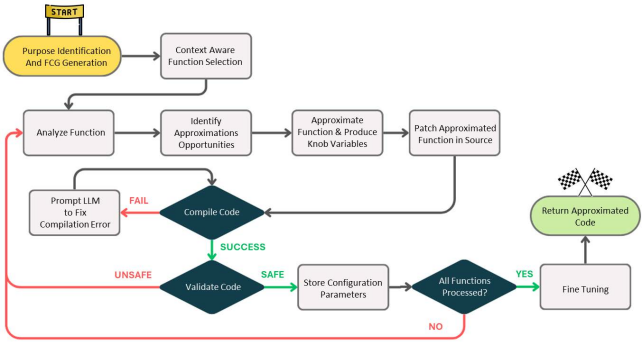


Figure 1: CheckMate workflow.

CoT prompting strategy collectively enhance CheckMate's ability to generate effective approximations while minimizing errors, for batteryless IoT applications through a one-click solution.

Figure 1 presents a detailed view of CheckMate 's workflow, illustrating the seamless interaction between its components. It outlines each step, from function identification to fine-tuning, showcasing the cohesive and systematic flow that underpins the approximation process.

3 Architecture and Implementation

CheckMate architecture enables seamless interaction among its core components, ultimately transforming user input into approximate code, as illustrated in Figure 2. In the following, we provide an in-depth exploration of each module's implementation.

```
void sobel_filter(double* image) {

   image = load_image(IMAGE_SIZE);
   double* new_image = [IMAGE_SIZE];

   for (int i = 0; i < IMAGE_SIZE; i++) {
      /* Applying Sobel Filter */
   }
   send_to_classifier(new_image);
}
```

Listing 1: Original sobel filter code.

3.1 User Input

CheckMate requires six essential inputs in JSON format, which users may modify: original application source-code, used to identify approximation opportunities; *input traces*, representing potential application inputs and energy harvesting traces, or, if unavailable, detailed input characteristics to simulate a trace; *accuracy class* (a_c), used to calculate output deviations (cf. Section 3.3 and Table1); and *error bound* (e_b), which defines acceptable deviation limits in the output. Additionally, knowledge of the *platform* and its architecture (e.g., Arm Cortex M or MSP430) aids in precise energy consumption estimation, while the *capacitor size* determines the minimum energy required to prevent non-progressive states, where the program fails to advance beyond a certain code segment [27]. An optional parameter, *LLM temperature*, can be adjusted, with a default low value (0–0.3), as this range minimizes the LLM's randomness and is generally preferred for coding tasks [16].

3.2 Approximation

After gathering the required inputs, CheckMate initiates the code approximation process. It begins by parsing the source code (Listing 1) and iteratively approximating functions selected by the LLM. Each approximated function undergoes verification to ensure it is error-free and safe for execution. We illustrate this workflow using the Sobel filter [35], a common edge detection algorithm in IoT, as a running example.

3.2.1 Code parsing. It consists of two primary steps: generating the FCG and extracting function definitions.

FCG generation: We employed Egypt [24] to generate the FCG, utilizing `gcc` to produce `.expanded` files for source code analysis and `Graphviz` for visualization. The resulting visualization file is processed to create an adjacency matrix representing the FCG.

To handle cyclic dependencies, cycles are arbitrarily removed [44]. After breaking these cycles, a topological sorting algorithm establishes the order for function approximation.

Function extraction: We use Language Server Protocol (LSP) [43] to accurately locate the line and character positions of function definitions within the original codebase. A custom Python script extracts these functions and stores their content in a JSON file.

Together, these processes provide the functions, their dependencies, and the order in which functions are approximated (from least to most dependent), forming a robust foundation for the next stages in the pipeline.

3.2.2 LLM interaction. We utilized LangChain [15], an open-source development framework, to streamline interactions with LLMs. LangChain's modular architecture supports integration with various LLMs, whether locally hosted or accessed through commercial APIs. For our evaluation, we employed OpenAI's GPT-4o, GPT-4o-mini, and Claude Sonnet 3.5 APIs [7, 46], while maintaining flexibility for future integration with other LLMs[4].

```
{
  code_summary: {
    "sobel_filter": "Applies sobel filter to image by
    ↪...",
    "load_image": "Loads and preprocess the image to
    ↪...",
    "send_to_classifier": "Uses BLE module to send...
    ↪",
  },
  target_functions: {
    "sobel_filter": "approximate",
    "load_image": "do not approximate",
    "send_to_classifier": "do not approximate",
  }
}
```

Listing 2: Function summaries and classification.

[4]We also intended to evaluate GPT-4o1; however, it remains in preview mode, and its API is not yet publicly available.

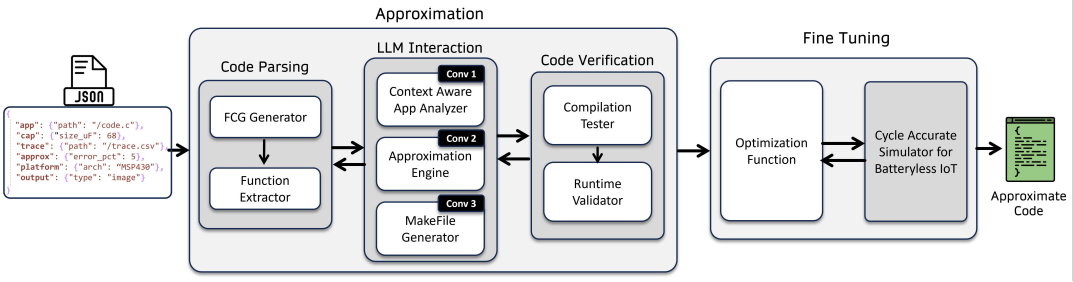


Figure 2: CheckMate architecture.

All interactions with the LLM start with a *system prompt* establishes the context, ensuring that the LLM's outputs align with the specific objectives and constraints of the task. For certain steps LLM begins with few-shot learning containing examples of original and approximated code for intermittent computing.

Function selection: Identifying functions for approximation is a critical step in our optimization process (**Conv 1**, Figure 2). This is achieved through a structured sequence of prompts.

Initially, the LLM is passed a system prompt detailing its purpose and overall objectives [A.1.6]. Next we present the entire application code to LLM to help it understand the overall objective of the application. It then generates concise summaries for each function, describing their purpose, inputs, and outputs [A.1.7]. Subsequently, each function is analyzed individually [A.1.8]. The LLM evaluates the role of each function within the application, determining which are suitable for approximation and identifying the potential benefits of such approximations. Finally, using a dedicated prompt, the LLM classifies each function as either "approximate" or "do not approximate" (Listing 2)[A.1.9]. This systematic approach allows the CheckMate to ensure that needless approximations and the resulting knobs are filtered out during the initial steps, as described next.

```
Reducing Image Size

There are two ways to approximate by resizing:

1. To approximate code, reducing the `IMAGE_SIZE`
     variable is perforated, as it decreases the number
     of iterations in the Sobel filter loop, `load_image`,
      and `send_to_classifier` functions. However, this
     may introduce an input error in `send_to_classifier`
      if the image size no longer aligns with the model's
      requirements. In contrast, perforating the Sobel
     filter loop only reduces iterations within that loop
      by skipping or truncating some pixels, keeping the
     overall image size constant, but it provides less
     cycle reduction compared to reducing the image size.

2. ...
```

Listing 3: LLM suggested approximation.

Approximating functions: Using the topological sort from the earlier steps and the list of functions selected for approximation, we filter the sort to include only the selected functions, which are processed sequentially. As shown in Figure 2, this step occurs in a separate conversation with a distinct purpose. Lacking any prior context, the LLM is reintroduced to the codebase using only the brief summary from the previous discussion. Additionally, a new system prompt is used to align with the conversation's specific objective [A.2.10].

The approximation process comprises two phases. In the first phase, the LLM is prompted to enumerate all possible approximations for the selected function and their potential benefits [A.2.11]. The LLM identifies suitable areas for approximation, highlights knobs, and returns a list of feasible code modifications (Listing 3) along with implementation details. As outlined in **Conv 2** (Figure 2), this phase guides the LLM to identify and plan the most appropriate approximations for the application.

```
void sobel_filter(int* image) {
    /* Knob Variables Declaration Start */
    int knob1 = 80;
    /* Knob Variables Declaration End */
    image = load_image(IMAGE_SIZE);
    int* new_image = [IMAGE_SIZE];

    for (int i = 0; i < (IMAGE_SIZE*knob1)/100; i++)
         {
        /* Applying Sobel Filter */
    }
    send_to_classifier(new_image);
}
```

Listing 4: Approximate code with knob variables.

In the second phase, the LLM is prompted to implement the selected approximations, including creating knobs [A.2.12]. The LLM also defines the range of these knobs to facilitate later fine-tuning (cf. Section 3.3). Depending on the potential approximation discussed in the previous response from the LLM, a few-shot example set may be provided to illustrate safe practices for applying specific types of approximations [A.2.14, A.2.15]. As shown in the example in Listing 4, the LLM applies loop perforation and introduces a knob. Up to this point, the interaction with LLM relies on plain text with embedded code snippets, but the variability in responses can complicate extracting the necessary information. To address this, a final prompt consolidates all data into a structured JSON object.

The resulting JSON object (Listing 5) provides a concise summary of the applied approximations, ready for further processing.

Makefile generation: CheckMate prompts the LLM to generate a makefile for automated compilation of the codebase. In this conversation (**Conv 3**, Figure 2), a system prompt is issued [A.3.16], followed by a user prompt listing the files in the codebase directory [A.3.17]. The LLM uses this information to produce the makefile, which incorporates all necessary compilation parameters, such as linker libraries and file names. The conversation history is saved for future reference, allowing for the generation of a new makefile if required in later steps.

```
{
    apx_code: "/*approximated code*/",
    knob_variables: ["knob1"],
    knob_ranges: [{"knob1": [20, 100]}],
    knob_increments: [{"knob1": "Integer"}]
}
```

Listing 5: LLM's output in JSON

3.2.3 Code verification. After a function is approximated, it is essential to verify that the modified code compiles and executes correctly. The approximated code is integrated into a copy of the codebase and prepared for testing.

Compilation: CheckMate compiles the patched codebase using the generated makefile. If compilation errors occur, the errors are logged and fed back to the LLM, which adjusts the makefile accordingly. This iterative process is repeated for a predefined number of attempts. If the code fails to compile after multiple iterations, CheckMate reverts to the beginning and applies an alternative approximation technique. Once the function successfully compiles, CheckMate transitions to the validation phase to evaluate runtime correctness. The automated generation of makefiles and iterative compilation significantly reduces the developer's workload, ensuring seamless integration of approximated code into the application's build process.

Validating Approximations: The final step in ensuring a safe approximation is runtime validation, which also confirms the safety of LLM-defined knob ranges. Validation starts by executing the approximated function using the upper and lower bounds of the knob values. If the initial executions are error-free, CheckMate performs a binary traversal across the knob variable's range to identify specific intervals where runtime errors may occur. If errors are detected, the search is refined to focus on intervals where the function executes without issues. The approximation is deemed unsafe and discarded if the function fails across all tested intervals.

3.3 Fine Tuning

Optimization Function: After successfully validating all approximations, the next step is to determine the most suitable knob values that balance output error and the number of power cycles by minimizing the following optimization metric (see Figure 3):

$$Optimization\ Metric = e_m + c_r \tag{1}$$

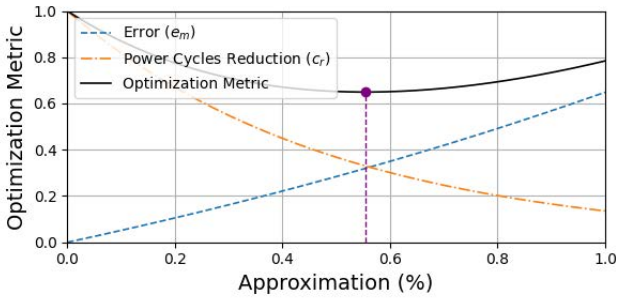


Figure 3: Minimizing Optimization Metric: The purple dot indicates the point where the optimization metric achieves its minimum value, representing the balance between reduced power cycles and acceptable output error.

where e_m quantifies the deviation between the original and approximated outputs:

$$e_m = \frac{|a_o - a_a|}{a_o} \tag{2}$$

where a_o and a_a denote the accuracy of the original and approximated outputs, respectively. e_m must remain below the error bound e_b. The accuracy of the output is evaluated across four key output data-types currently supported by CheckMate, including numeric, text, image, and boolean (see Table 1), enabling broad applicability to IoT use cases like environmental sensing and edge machine learning. Nonetheless, adding a new output type only requires adding a new error calculation function.

The reduction in power cycles, c_r, is defined as:

$$c_r = \frac{c_a}{c_o} \tag{3}$$

where c_o and c_a represent power cycles in the original and approximated programs, respectively. Typically, $0 < c_r \leq 1$.

CheckMate employs Bayesian optimization [57] to minimize the optimization metric leveraging a probabilistic model (e.g., Gaussian Process). Implemented using the `gp_minimize` method from the scikit-optimize (`skopt`) library [26], this approach ensures effective fine-tuning of knob variables. Traditional gradient-based methods, like gradient descent, are unsuitable due to the non-differentiable, computationally intensive nature of our objective function. Alternatives like brute-force and evolutionary algorithms are computationally prohibitive for time-sensitive evaluations.

Simulation: Both e_m and c_r are evaluated using the *Fused* simulator [60], which models power consumption and energy storage, and emulates checkpointing and system reboots, vital for our evaluations. Fused was chosen over emulators like *Renode* [54] and *MSPSim* [21] for its extensive support for intermittent computing and compatibility with multiple architectures, including Cortex-M and MSP430.

4 Evaluation

Our evaluation encompasses a detailed performance study, baseline comparisons, testbed experiments, and a small-scale user study.

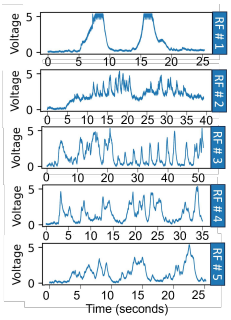


Figure 4: RF Traces

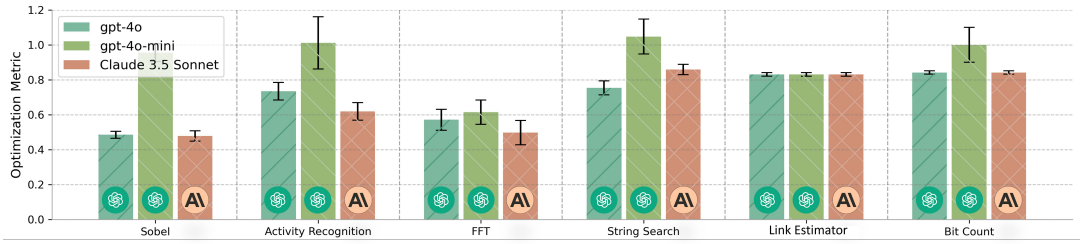


Figure 5: Optimization metrics for each LLM. Depicting performance of GPT-4o (right bar), GPT-4o Mini (middle bar), and Claude 3.5 Sonnet (left bar) across six applications. The lower values of the optimization metric (e_m+c_r) indicate better performance.

Setting: For our evaluation, we employed ManagedState [59] within Fused as the checkpointing mechanism for retaining program state across power cycles. Reducing checkpointing overhead, though, is an orthogonal problem, not the primary focus of our study. CheckMate's benefits are expected to generalize across various checkpointing solutions.

For buffering harvested energy, our experiments used three capacitor sizes: the smallest required for each workload, as established in prior studies [14, 18], and two subsequently larger standard sizes. To emulate realistic energy conditions, we incorporated five distinct open-source RF energy traces (Figure 4), commonly employed in previous research [14, 51, 52]. We set a 30% upper bound on output error, informed by prior research [48], to define the search space for Bayesian optimization and enable effective fine-tuning of knobs. A user, however, can modify this value arbitrarily to align with their specific application requirements. The Bayesian optimization process, conducted over 150 iterations as shown in Figure 6, balanced computational overhead with achieving solutions close to the global minimum. Further iterations provided only marginal gains in energy efficiency or accuracy, ensuring a practical trade-off between computation time and solution quality.

Applications: We evaluated CheckMate on six diverse applications, widely employed in intermittent computing evaluations [2, 14, 41, 52, 64], that offer diverse opportunities for approximation.

(1) ***Sobel***: An image processing algorithm for edge detection, highlighting regions of high spatial gradient. Commonly used in IoT applications such as real-time image processing on edge devices [34].
(2) ***Activity Recognition***: Classifies a user's physical state (e.g., walking, running, stationary) using accelerometer data. Widely employed in IoT system benchmarks [23, 53].
(3) ***Fast-Fourier Transform (FFT)***: Converts signals from the time to frequency domain, commonly used in IoT applications like real-time audio processing [8] and vibration analysis [61].
(4) ***Boyer-Moore String Search***: Efficiently locates substrings within a main string. Used in IoT applications for text processing and pattern recognition [20, 38].
(5) ***Link Estimator***: Evaluates the strength and reliability of wireless communication links, widely used by standard routing protocols like RPL for route establishment [62].
(6) ***Bitcount***: Counts the number of set bits in a binary number, a key operation in IoT applications like data compression [58].

Table 1: Output types and accuracy classes.

Program Output	Accuracy Class
Numeric Values	Raw Absolute Error Normalized R-squared
Text	1 - Word Error Rate
Image	1 - Pixel Error Rate Structural Similarity Index
Boolean	F1-Score

Table 2: Application size. The reported sizes exclude storage used by the checkpointing strategy, which can add up to 58 kB of Flash. Token counts represent the number of tokens required to tokenize the codebase.

Application	Total Flash Usage	Total RAM Usage	Token Count	Line of Code
Sobel	9.0 kB	3712 B	1002	123
Activity Recognition	1200 B	274 B	2213	307
FFT	11413 B	26 B	911	119
String Search	4324 B	942 B	689	94
Link Estimator	9600 B	2516 B	403	78
Bitcount	574 B	10 B	489	80

As shown in Table 2, these benchmarks encompass a wide range of programming structures and varying code complexities, enabling a thorough evaluation of CheckMate's versatility and effectiveness across a spectrum of IoT applications. For instance, the Sobel filter involves computationally intensive image processing, while Activity Recognition employs classification with limited adjustable parameters. The FFT focuses on mathematical computations optimized for low-power microcontrollers, whereas the link estimator handles lightweight network statistics. The Boyer-Moore String Search addresses complex string matching with specific data dependencies, and Bitcount involves bit manipulation with simple control structures.

Key findings: Our findings are:

- Claude 3.5 Sonnet is identified as the most effective LLM for CheckMate, consistently delivering superior performance across

applications while ensuring adaptability to future advancements in LLM technologies (Section 4.1).
- CheckMate demonstrates its effectiveness in optimizing energy efficiency across diverse IoT applications by achieving substantial reductions in power cycles (up to 60%) while maintaining acceptable error margins in the output. (Section 4.2).
- CheckMate outperforms existing semi-automated frameworks like ACCEPT by achieving higher speedups and comparable or lower errors, all while operating fully autonomously, demonstrating its superiority and user-friendliness (Section 4.3).
- Evaluation on a testbed replaying energy harvesting traces and running the program on hardware demonstrates consistent performance, validating our simulation results (Section 4.4).
- A user study comprising 17 participants with varying programming expertise levels successfully used CheckMate to approximate IoT applications, highlighting its accessibility, ease of use, and potential for adoption in diverse user scenarios without requiring specialized knowledge (Section 4.5).

4.1 Results → LLM Evaluation

As a preliminary step, we identified the most suitable LLM for CheckMate, as it directly affects the quality of approximations. We evaluated three commercially available LLMs: OpenAI's GPT-4o (snapshot 2024-08-06), GPT-4o Mini (snapshot 2024-07-18), and Anthropic's Claude 3.5 Sonnet (snapshot 2024-10-22). Each model was rigorously tested across six applications, with experiments repeated 20 times to ensure statistical robustness. The aggregated results, including error bars representing confidence intervals, are shown in Figure 5.

Our analysis shows that Claude 3.5 Sonnet consistently outperforms the other models, particularly in larger applications such as FFT. GPT-4o Mini exhibits greater variability, likely due to its limited computational resources, which can lead to suboptimal approximations. Based on these findings, we selected Claude 3.5 Sonnet as the primary LLM for CheckMate, ensuring reliable and accurate approximations in subsequent evaluations. As advancements in LLMs continue at a rapid pace, CheckMate is compatible with emerging models.

4.2 Results → Performance

We conducted comprehensive performance benchmarks to evaluate CheckMate 's effectiveness under varying energy constraints.

Sobel filter: We evaluated the Sobel filter implementation from the ACCEPT repository [56], using the Structural Similarity Index Measure (SSIM), categorized under the image output type in Table 1, to assess output fidelity. CheckMate automatically identified two approximation opportunities in the Sobel filter: precision scaling and loop perforation. Precision scaling reduced numerical precision from double-precision floating-point to integer values, while loop perforation was applied to the input image and filtering loops.

During optimization, the Sobel filter's knobs demonstrated high sensitivity, with only a narrow range yielding acceptable output errors. This sensitivity resulted in multiple iterations yielding a 100% error, as any error surpassing the predefined threshold was capped at the maximum allowable value. Figure 6 illustrates this, showing a distinct gap between two clusters in the optimization metric: the upper cluster reflects unacceptable errors, while the lower cluster includes iterations meeting the error criteria.

Despite these challenges, the optimization process delivered substantial improvements, , as shown in Figure 8a. Precision scaling alone reduced power cycles by 49% with minimal SSIM loss. Combining precision scaling with loop perforation further decreased power cycles from 59 to 23, achieving an overall reduction of approximately 60%. These results were obtained with the smallest capacitor size of 68 μF, the minimum required for Sobel filter operation. Similar power cycle reductions and output error rates were observed with larger capacitor sizes. The visual effects of these approximations are depicted in Figure 7. Precision scaling retains less information, causing pixels to shift to darker values due to approximate calculations. With input image truncation from loop perforation, two black bars appear, marking sections where the filter was not applied.

Activity Recognition: We use ACCEPT's implementation of activity recognition [56], using the F1-score as the primary metric to assess classification accuracy, chosen for its balanced consideration of precision and recall.

Through automated analysis, CheckMate identified two effective approximation techniques: loop perforation and feature approximation. Loop perforation involved selectively skipping certain data points to reduce computational load, while feature approximation simplified calculations of key statistical features, such as mean and standard deviation, during feature extraction. Combining these techniques achieved a 28% reduction in power cycles (from 14 to 9) while maintaining high classification accuracy (see Figure 8b).

The optimization process converged quickly, reaching its lowest metric at the 53rd iteration with an output error of only 1%, attributed to the limited number of adjustable parameters. Figure 6 illustrates the fine-tuning of activity recognition during, demonstrating the effectiveness of our approximation strategies.

FFT: We evaluated the Cooley-Tukey FFT algorithm [9] using Root Mean Square Error (RMSE) as the accuracy metric, and minimum capacitor size of 22 μF.

CheckMate identified five approximation strategies: Beyond optimizing the FFT algorithm itself, the LLM identified approximation opportunities in the Taylor series expansions for exponential and trigonometric functions, frequently used in FFT calculations. By adjusting the iteration count (i.e., the n-th term) in the Taylor series for cos, sin, and exp functions—and assigning a knob to control each—CheckMate managed to achieve a trade-off between output error and power cycle reduction. Precision scaling was also applied, switching from double-precision to single-precision floating-point data types. These techniques achieved a 51% reduction in power cycles with merely 7.5% RMSE, as shown in Figure 8c.

Boyer-Moore String Search: We employed Boyer-Moore string search algorithm from the MiBench2 benchmark suite [25] using F1-score as the accuracy metric and a capacitor size of 68 μF.

CheckMate achieved a 31% reduction in power cycles with only a 6.6% error in the output value on the smallest capacitor, as shown in Figure 8d. Importantly, this application demonstrates how LLMs can effectively interpret application context to apply domain-specific optimizations. During the function selection phase, the LLM identified

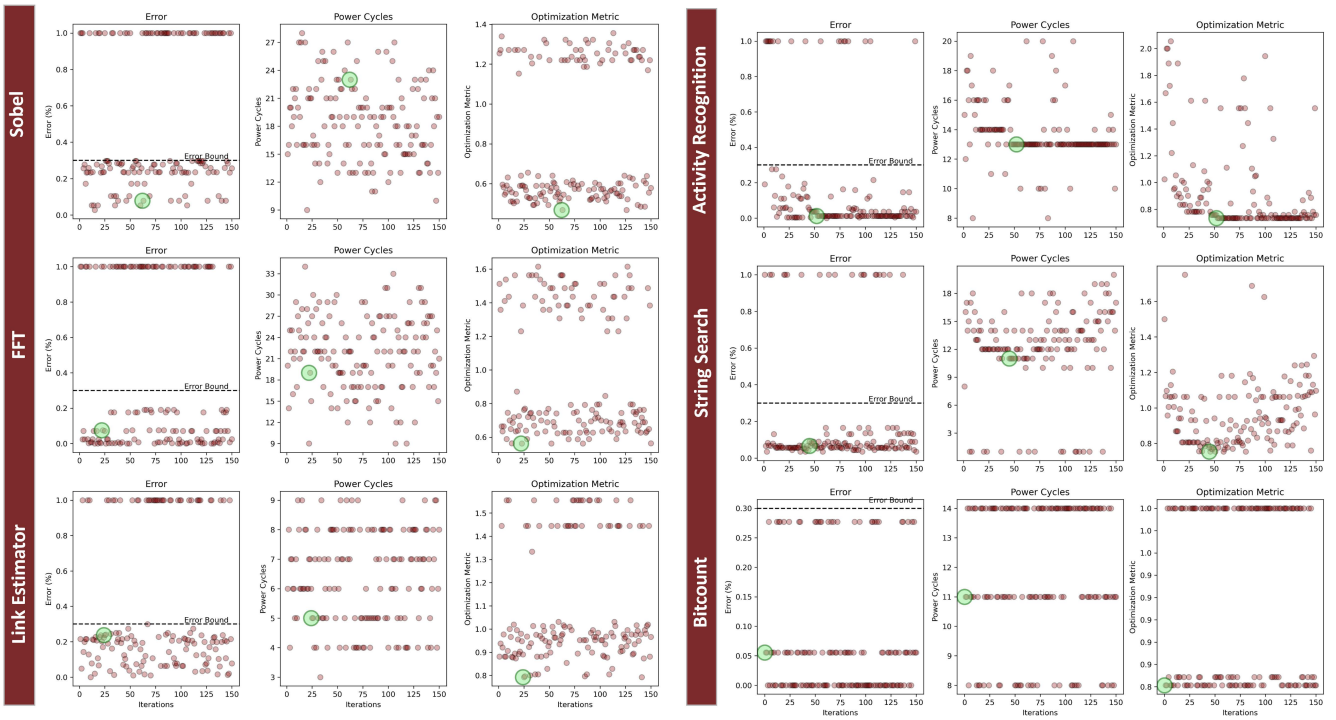


Figure 6: Bayesian optimization results for six applications. Each row represents an application, featuring subplots for output error (left), number of power cycles (middle), and optimization metric (right) across iterations. Green circles highlight the iteration at which the minimum optimization metric is reached, indicating the optimal balance between output error (%) and the number of power cycles for each application.

Table 3: Reduction in power cycles(%) and output error rates (%) on remaining RF energy traces.

Trace	Sobel		AR		FFT		String Search		Link Estimator		Bit Count	
	Red. (%)	Err. (%)	Red. (%)	Err. (%)	Red. (%)	Err. (%)	Red. (%)	Err. (%)	Red. (%)	Err. (%)	Red. (%)	Err. (%)
RF #1	63.16	17.37	40.10	4.87	47.67	0.01	33.33	2.48	36.36	21.50	36.36	27.72
RF #3	55.77	7.25	28.57	3.25	58.00	1.03	24.00	0.00	45.45	24.50	18.18	5.56
RF #4	54.00	11.92	35.29	3.25	51.23	0.06	25.00	2.48	41.67	21.50	41.67	27.72
RF #5	56.52	17.37	23.08	8.12	52.33	0.01	25.00	0.00	38.46	21.50	21.43	5.56

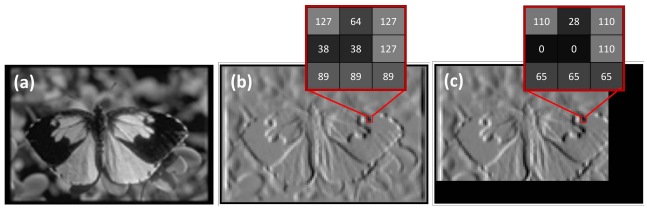
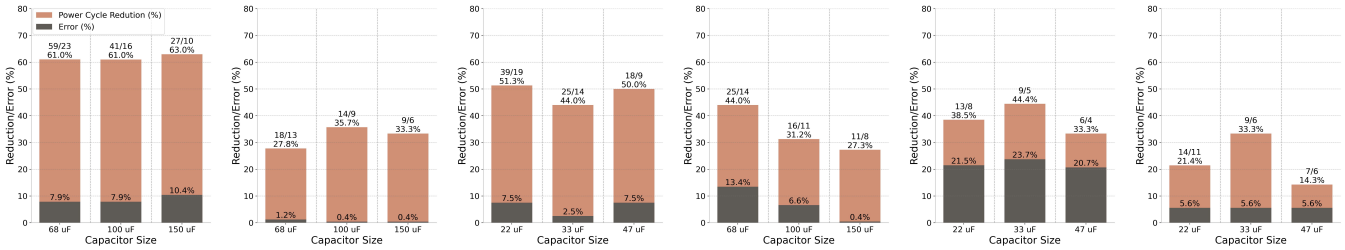


Figure 7: Original vs approximated Sobel. (a) The original image. (b) Edge detection using the original code. (c) Edge detection using the approximated code.

an opportunity to approximate the initialization step in the Boyer-Moore algorithm. Specifically, it observed that the input traces did not cover the full ASCII range, reducing the `bad character table` to include only frequently encountered characters.

We performed an additional test to validate this observation by withholding the input trace from the LLM. Without the input trace, the LLM skipped the `bad character table` approximation and relied solely on perforating the main search loop, resulting in a consistent 6.6% error and a 10% reduction in power cycles. This experiment clarified the consistently low error percentages seen during the fine-tuning process (Figure 6), as adjusting the knob that controlled the `bad character table`'s size did not increase error but impacted power cycles due to the larger table's storage and checkpointing requirements. These findings underscore CheckMate's capacity to harness LLMs for intelligent, context-sensitive optimizations.



(a) Sobel **(b) Activity Rec.** **(c) FFT** **(d) String Search** **(e) Link Estimator** **(f) Bitcount**

Figure 8: Performance results (RF#2 trace across three capacitor sizes). The darker bars represent the output error percentage, while the lighter bars illustrate the reduction in power cycles. Values above each bar indicate the original and approximated cycle counts, along with the corresponding percentage reduction.

Link Estimator: We also evaluated CheckMate on a simple link quality estimator that applies an exponentially weighted moving (EWMA) average to packet reception rates. This estimator is an integral part of several RPL implementations [45]

We evaluated the accuracy of the estimator using percentage absolute error. Given its straightforward codebase, this application offered a single approximation opportunity: loop perforation on link's running history of packet reception rates. As shown in Figure 8e, CheckMate achieved a 38% reduction in power cycles with a 21% error EWMA. Like the Sobel filter, the fine-tuning in Figure 6 displayed two clusters: one with errors beyond the acceptable threshold and the other within acceptable bounds. Unlike Sobel and other applications, however, this application exhibited a narrower range of power cycles, likely due to its simple computations.

Bitcount: We adapted the bitcount implementation from the MiBench2 suite [25] to evaluate CheckMate using 100 predefined random numbers. Due to the simplicity of this application, CheckMate identified only one viable approximation: perforating the main counting loop to include only the n most significant bits. This approximation also had a small search space, resulting in a low-variance distribution observed in Figure 6. CheckMate achieved a 21.4% reduction in power cycles with an output error of less than 6% (Figure 8f).

Results summary: Table 3 summarizes our results across the remaining RF energy traces for the smallest capacitor size demonstrating that CheckMate retains its performance benefits across all traces and applications. Minor variations arise due to differing energy availability patterns in each trace, influencing the number of power cycles and subsequently affecting the optimization metric.

Benchmarking CheckMate across six diverse applications demonstrates its capability to generate energy-efficient, approximated code with minimal loss in output quality. Among the evaluated applications, the Sobel filter exhibited the most significant performance gains, underscoring CheckMate's effectiveness in optimizing resource-intensive image processing tasks. The activity recognition algorithm showcased rapid convergence with minimal error, highlighting efficiency in systems with limited adjustable parameters. The FFT evaluation illustrates CheckMate's strength in optimizing computationally intensive mathematical functions, achieving substantial energy savings. The link estimation algorithm exemplifies CheckMate's adaptability to lightweight applications with small power budgets. Lastly, the string search task emphasizes the value of context-aware approximation, as CheckMate leverages input characteristics to enhance efficiency.

Table 4: CheckMate vs ACCEPT

Applications	ACCEPT		CheckMate	
	Speedup	Error	Speedup	Error
Sobel	2.0×	≈ 26.7%	2.6×	≈ 8.7%
Activity Recognition	1.5×	≈ 0.1 %	1.6×	≈ 0.6%

4.3 Results → Baseline Comparison

To evaluate CheckMate's performance relative to existing frameworks, we compared it with ACCEPT [56], a semi-automated tool requiring expert knowledge for identifying approximation opportunities within the code. In contrast, CheckMate is fully automated, requiring no specialized user input.

We tested CheckMate on the two embedded applications, Sobel and Activity Recognition, from the ACCEPT repository, using the speedup metric—defined as the ratio of clock cycles in the original to approximated code—to measure processing time reduction, ensuring comparability with ACCEPT's results.

We selected these two applications to ensure a direct and fair comparison with ACCEPT. The developers of ACCEPT evaluated and reported results exclusively on these two embedded system applications, making them the only available baseline for comparison. Expanding our benchmark set beyond these applications would have introduced methodological inconsistencies, reducing the validity of direct performance comparisons.

As shown in Table 4, CheckMate achieved significantly higher speedup for the Sobel application while maintaining greater accuracy. For Activity Recognition, CheckMate delivered comparable performance with a slightly higher error rate without any user intervention. These results demonstrate that CheckMate can match or surpass the efficiency of semi-automated, expert-driven frameworks while offering a single-click solution.

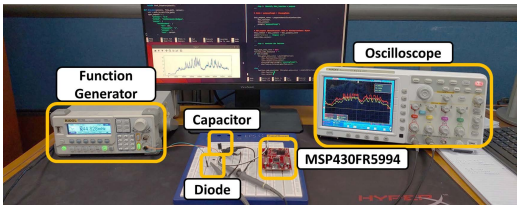


Figure 9: Testbed setup.

Table 5: Simulation vs testbed

Applications	Sim Red.(%)	Testbed Red.(%)	Delta Diff.(%)
Sobel	61.0	66.6	+5.6
Activity Rec	35.7	33.3	-2.4
FFT	51.3	49.4	+1.9
String Search	31.2	30.0	-1.2
Link Estimator	38.5	40.2	-1.7
Bit Count	21.4	22.0	-0.6

4.4 Testbed Evaluation

A testbed evaluation is crucial for assessing CheckMate's reliability beyond simulated environments. Our testbed (Figure 9) consists of a function generator (RIGOL DG1022 [12]) for replaying energy traces and the MSP430FR5994 Launchpad [28] as the target platform.

The function generator, featuring a 14-bit digital-to-analog converter (DAC), scales voltage values in RF traces to integer values between 0 and 2^{14}. This scaling is performed on a laptop and programmed into the generator using the Python `pyVISA` library [49], which issues VISA commands for precise read-write operations. This approach eliminates the need for additional amplification circuitry, as the generator can directly supply sufficient current and voltage to the energy harvesting system. The energy harvesting setup includes a capacitor, diode, and the MSP430. The capacitor stores energy from the generator's voltage signal, powering the MCU. The diode prevents reverse discharge, blocking current flow when the generator outputs a voltage lower than the capacitor's stored charge. Voltage levels from both the energy trace and capacitor are monitored via an oscilloscope to ensure accurate energy trace replication. To trace the power cycles required for program execution, we saved the count in a dedicated FRAM region on the MSP430.

We conducted experiments using five distinct RF energy traces [51] and present detailed results for the RF#2 trace to compare these results with the ones presented in Figure 8. Table 5 illustrates a strong correlation between simulation and real-world results.

We only report reductions in power cycles, as the error remains consistent for the approximated code in both simulation and hardware evaluations. Minor fluctuations in percentage reductions between the testbed and simulation results arise from real-world factors such as effective resistance, capacitor leakage, and other non-ideal parameters not fully modeled by the simulator. These discrepancies are more pronounced in applications with extended runtimes and larger codebases.

4.5 Small-scale user study

We conducted a user study involving seventeen computer science students to evaluate the user experience of CheckMate. Our key findings reveal that CheckMate offers two major benefits: 1) it significantly reduces the time required for performing approximations, and 2) it reduces the chances of higher errors in the output compared to manual methods.

Methodology: Participants were first introduced to approximate computing concepts and provided with documentation on writing approximate programs in C. They had 30 minutes to familiarize themselves with the material. To bridge theory with practice, each student was then walked through a complete application, showcasing effective application of approximation techniques. Following this, participants tackled three additional applications, identical to our benchmarks, applying approximation techniques over the course of one hour using standard C practices. Throughout these exercises, a cycle-accurate simulator was at their disposal, enabling real-time evaluation of their approximations.

After completing the manual tasks, participants received documentation on CheckMate and a brief tutorial on its usage. They then reattempted the same programming challenges using CheckMate under identical conditions. Upon completion, participants filled out a survey assessing CheckMate's usability and effectiveness.

Participants: The study comprised computer science students from both junior and senior years of our undergrad program. These participants brought a diverse range of experiences, with three to seven years of formal computing education[5]. This varied background ensured a comprehensive evaluation of CheckMate across different academic levels and skill sets.

Results: Using CheckMate significantly reduced the average time required for performing approximations, decreasing from 13.93 minutes (manual) to 2.56 minutes per task. Moreover, manual approximations yielded suboptimal optimization metrics compared to CheckMate. Achieving a comparable optimization metric manually would require an indeterminate amount of time, making it impractical for evaluation within the study's timing constraints. Manual approximations produced output error rates ranging from 51% to 82% with inconsistent power cycle reductions[6]. In contrast, CheckMate consistently delivered superior results, as detailed in Section 4.2. Participants reported that CheckMate was intuitive and user-friendly, quickly mastering its features. A significant majority (93%) indicated they would strongly recommend CheckMate for implementing approximations in intermittent computing applications.

5 Discussion and Limitations

The presented solution approach also introduces certain limitations and challenges related to the performance and practicability of CheckMate. In this section, we highlight some of these shortcomings and discuss potential solutions that users may adopt.

Approximation overhead: Several approximations introduce computational overhead. For example, a loop perforation approximation

[5]Some students had extensively studied programming during high school.
[6]No correlation was observed between output error rates and power cycle reductions.

may add extra floating-point arithmetic operations, which contribute to the overall computation of the application. On its own, this overhead may not significantly alter the number of checkpoints required for the application to run to completion. However, when a large number of knobs are created by the LLM, it is possible for these knobs to collectively generate a significant overhead.

This overhead becomes particularly significant when many of the created knobs are 'tuned out' by the Bayesian optimizer. A knob is considered tuned out when its setting ensures that the corresponding section of code behaves identically to the original application code. Consequently, tuned-out knobs introduce unnecessary computational overhead.

Much of this issue is mitigated by compilers, which automatically eliminate redundant code. However, there are instances where the compiler fails to remove such overhead. These typically arise when the LLM provides knob ranges 5 that lack a value capable of making the program behave exactly as the original code. In such cases, the Bayesian optimizer selects the knob value closest to "zero" (i.e., the setting most similar to the original program operation), but the compiler remains unable to eliminate the associated overhead.

Privacy concerns: Using commercial APIs for testing and evaluation of CheckMate can raise privacy concerns for developers operating in proprietary or confidential environments. To address this, CheckMate is designed to seamlessly integrate with any LLM API via LangChain, enabling it to operate with locally deployed LLMs. With the rapid advancements in open-source LLMs, such as DeepSeek's [19], many of which have reached performance levels comparable to commercial alternatives while requiring less computational power and budget, it is reasonable to assert that CheckMate would deliver similar performance when leveraging these open-source models.

LLM Specifications and Application Size Dynamics: The relationship between application size and the number of model parameters is a critical determinant of performance. As shown in Figure 5, CheckMate's effectiveness declines significantly when using smaller LLMs. For instance, while both GPT-4o and GPT-4o Mini share the same context window size, they differ in parameter count. The smaller GPT-4o Mini struggles with larger and more complex applications, exhibiting noticeable performance degradation. Additionally, it fails to capture some of the optimal approximations identified by its larger counterpart.

Another key factor is the relationship between application size and context window length. Larger applications require a longer context window for effective processing. To ensure broad generalizability across various LLMs while maintaining efficiency, CheckMate was designed to fit within a 16k-token context window[7].

6 Related Work

Related efforts can be divided into two broad categories.

Application-specific approximations: Approximation techniques in intermittent computing are often limited to specific applications. Bambusi et al. [10] introduced approximate intermittent computing, trading computational accuracy for reduced state maintenance overhead. Applied to human activity recognition and image processing, their approach achieved significant throughput gains with minimal accuracy loss. In deep learning on intermittently powered devices, Islam and Nirjon [29] developed Zygarde, which adjusts neural network computations based on energy availability for timely processing. Lin et al. [37] proposed intermittent-aware neural network pruning, reducing computational load without significant loss in accuracy. Barjami et al. [11] demonstrated that slight accuracy reductions can substantially boost throughput in intermittent inference, highlighting the potential of controlled approximation. In [50], the authors explore algorithmic-level approximation techniques such as loop perforation and synaptic pruning specifically for neural networks. Ganesan et al. [22] presented the "What's Next" architecture, integrating approximate computing into intermittent systems using anytime algorithms and hardware modifications. Evaluated on a 2D convolution application, it effectively maintained output quality under energy constraints.

Approximation frameworks: Closer to our work are solutions that provide frameworks for applying approximations. ACCEPT [56] provides a comprehensive framework that integrates compiler analysis and auto-tuning to automate program relaxation while adhering to safety constraints. However, it requires extensive manual tuning and expert intervention to achieve acceptable results. Puppeteer [47] allows developers to annotate code regions to quantify the sensitivity of application outputs to approximation errors. While this provides some control, it places a significant burden on developers to correctly identify and annotate sensitive code regions. Rumba [36] tackles the issue of managing error in approximations through continuous verifications and adjustments. Yet, it adopts a one-size-fits-all strategy that lacks the flexibility needed to balance energy efficiency and computational accuracy in highly diverse applications.

CheckMate is a software-only framework designed to address distinct challenges of intermittent computing on batteryless IoT. Unlike generic approximation tools, CheckMate uniquely balances power cycles and output errors, optimizing energy efficiency while ensuring desired output fidelity.

7 Conclusion

We presented CheckMate, a novel framework that leverages LLMs to automate approximate intermittent computing. CheckMate integrates LLM-driven approximations with robust validation processes, dynamic knobs, and Bayesian optimization to effectively balance energy efficiency and computational accuracy with minimal developer input. Evaluations across diverse IoT applications demonstrated substantial reductions in power cycles while maintaining output errors within user-defined bounds. Real-world testbed results aligned closely with simulation predictions, underscoring CheckMate's reliability. These advancements position CheckMate as a foundational step towards enabling scalable and user-friendly, automated approximation solutions for batteryless IoT environments.

8 Acknowledgments

This work was supported by the Higher Education Commission of Pakistan (HEC) under the National Research Program for Universities (NRPU) grant (award number 20-15968/NRPU/R&D/HEC/2021-2020).

[7] Different LLMs use varying tokenization strategies.

References

[1] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, et al. 2020. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 368–381.

[2] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 70–81.

[3] Saad Ahmed, Naveed Anwar Bhatti, Martina Brachmann, et al. 2021. A survey on program-state retention for transiently-powered systems. *Journal of Systems Architecture* 115 (2021), 102013.

[4] Saad Ahmed, Bashima Islam, Kasim Sinan Yildirim, Marco Zimmerling, Przemysław Pawełczak, Muhammad Hamad Alizai, Brandon Lucia, Luca Mottola, Jacob Sorber, and Josiah Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* 67, 3 (2024), 64–73.

[5] Jawaher Alharbi and Arshad Jhumka. 2023. Checkpointing in Transiently Powered IoT Networks. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*. 316–317.

[6] Jawaher Alharbi, Arshad Jhumka, and Daniele Palossi. 2023. To Checkpoint or Not to Checkpoint: That is the Question *(EWSN)*. ACM, New York, NY, USA, 124–131.

[7] Anthropic. [n. d.]. Claude 3.5: Introducing a New Era of AI Assistance. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2024-11-07.

[8] Mattia Antonini, Massimo Vecchio, Fabio Antonelli, Pietro Ducange, and Charith Perera. 2018. Smart audio sensors in the internet of things edge for anomaly detection. *IEEE Access* 6 (2018), 67594–67610.

[9] Brendan Ashworth. 2019. Fast Fourier Transform. https://github.com/brendanashworth/fft-small.

[10] Fulvio Bambusi, Francesco Cerizzi, Yamin Lee, et al. 2022. The case for approximate intermittent computing. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 463–476.

[11] Rei Barjami, Antonio Miele, and Luca Mottola. 2024. Intermittent Inference: Trading a 1% Accuracy Loss for a 1.9x Throughput Speedup. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems* (Hangzhou, China) *(SenSys '24)*. Association for Computing Machinery, New York, NY, USA, 647–660. https://doi.org/10.1145/3666025.3699364

[12] Batronix. 2024. Rigol DG1022 Waveform Generator. https://www.batronix.com/shop/waveform-generator/Rigol-DG1022.html. Accessed: 2024-11-12.

[13] Naveed Bhatti, Luca Mottola, et al. 2016. Efficient state retention for transiently-powered embedded sensing. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*. 137–148.

[14] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*. 209–219.

[15] Harrison Chase. 2024. LangChain Documentation. https://www.langchain.com/. Accessed: 2024-06-28.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[17] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 514–530.

[18] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 767–781.

[19] DeepSeek-AI. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL]

[20] Menachem Domb. 2021. A framework of signature-matching-algorithms for IoT intrusion detection. In *Proceedings of the Future Technologies Conference (FTC) 2020, Volume 3*. Springer, 889–898.

[21] Joakim Eriksson and et al. 2007. Poster Abstract: MSPsim–an Extensible Simulator for MSP430-equipped Sensor Boards. (01 2007).

[22] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. 2019. The what's next intermittent computing architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 211–223.

[23] Alessandro Ghibellini, Luciano Bononi, and Marco Di Felice. 2022. Intelligence at the iot edge: Activity recognition with low-power microcontrollers and convolutional neural networks. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 707–710.

[24] Andreas Gustafsson. unknown. Egypt - Create Call Graph from GCC RTL Dump. https://www.gson.org/egypt/egypt.html. Retrieved June 23, 2024.

[25] M.R. Guthaus and et al. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[26] Tim Head, MechCoder, Gilles Louppe, et al. 2018. *scikit-optimize/scikit-optimize: v0.5.2*. https://doi.org/10.5281/zenodo.1207017

[27] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–13.

[28] Texas Instruments. [n. d.]. MSP430 MCUs - Ultra-low-power microcontrollers for embedded development. https://www.ti.com/design-development/embedded-development/msp430-mcus.html. Accessed: 2024-11-07.

[29] Bashima Islam and Shahriar Nirjon. 2019. Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems. *arXiv preprint arXiv:1905.03854* (2019).

[30] Kashif Javed, Naveed Anwar Bhatti, and Mohammad Imran. 2023. MOPTIC-SM: Sleep mode-enabled multi-optimized intermittent computing for transiently powered systems. *Journal of Systems Architecture* 137 (2023), 102850.

[31] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, et al. 2015. QuickRecall: A HW/SW approach for computing across power cycles in transiently powered computers. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 12, 1 (2015), 1–19.

[32] Seunghyeok Jeon, Yonghun Choi, Yeonwoo Cho, et al. 2023. HarvNet: Resource-Optimized Operation of Multi-Exit Deep Neural Networks on Energy Harvesting Devices. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services* (Helsinki, Finland) *(MobiSys '23)*. Association for Computing Machinery, New York, NY, USA, 42–55. https://doi.org/10.1145/3581791.3596845

[33] Min Jia, Edwin Hsing-Mean Sha, Qingfeng Zhuge, et al. 2022. Transient computing for energy harvesting systems: A survey. *Journal of Systems Architecture* 132 (2022), 102743.

[34] Rajeev Joshi, Md Adnan Zaman, and Srinivas Katkoori. 2022. Fast Sobel edge detection for IoT edge devices. *SN Computer Science* 3, 4 (2022), 302.

[35] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. 1988. Design of an image edge detection filter using the Sobel operator. *IEEE Journal of solid-state circuits* 23, 2 (1988), 358–367.

[36] Daya S Khudia, Babak Zamirai, Mehrzad Samadi, et al. 2015. Rumba: An online quality management system for approximate computing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 554–566. https://doi.org/10.1145/2749469.2750371

[37] Chih-Chia Lin, Chia-Yin Liu, Chih-Hsuan Yen, et al. 2023. Intermittent-aware neural network pruning. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[38] Chunwei Liu. 2022. *Fast and Effective Compression for IoT Systems*. Ph. D. Dissertation. The University of Chicago.

[39] Brandon Lucia, Vignesh Balaji, Alexei Colin, et al. 2017. Intermittent computing: Challenges and opportunities. *2nd Summit on Advances in Programming Languages (SNAPL 2017)* (2017).

[40] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[41] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 129–144.

[42] Andrea Maioli and Luca Mottola. 2021. Alfred: Virtual memory for intermittent computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 261–273.

[43] Microsoft. 2024. Language Server Protocol Documentation. https://microsoft.github.io/language-server-protocol/. Accessed: 2024-06-28.

[44] Nausheen Mohammed, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2024. Enabling Memory Safety of C Programs using LLMs. *arXiv preprint arXiv:2404.01096* (2024).

[45] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. 2022. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX* 18 (2022), 101089.

[46] OpenAI. 2024. OpenAI Platform Documentation: GPT-4 Turbo and GPT-4. https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4. Accessed: 2024-06-28.

[47] Konstantinos Parasyris, James Diffenderfer, Harshitha Menon, Ignacio Laguna, Jackson Vanover, Ryan Vogt, and Daniel Osei-Kuffuor. 2022. Approximate Computing Through the Lens of Uncertainty Quantification. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41404.2022.00072

[48] DRAGOŞ-ŞTEFAN PERJU. 2019. Applying Memoization as an Approximate Computing Method for Transiently Powered Systems. (2019).

[49] PyVISA Contributors. 2024. PyVISA Documentation. https://pyvisa.readthedocs.io/en/latest/. Accessed: 2024-11-12.

[50] Jeya Prakash R and Anju S Pillai. 2022. Enhancing Energy Efficiency of Intensive Computing Applications using Approximate Computing. In *2022 3rd International Conference on Electronics and Sustainable Communication Systems (ICESC)*. 1231–1236. https://doi.org/10.1109/ICESC54411.2022.9885429

[51] Benjamin Ransford. [n. d.]. Mspsim/traces at mementos · Ransford/mspsim. https://github.com/ransford/mspsim/tree/mementos/traces

[52] Benjamin Ransford and et al. 2011. Mementos: system support for long-running computation on RFID-scale devices. In *ASPLOS* (California, USA). New York, NY, USA, 159–170. https://doi.org/10.1145/1950365.1950386

[53] Nafiul Rashid, Berken Utku Demirel, and Mohammad Abdullah Al Faruque. 2022. AHAR: Adaptive CNN for energy-efficient human activity recognition in low-power edge devices. *IEEE Internet of Things Journal* 9, 15 (2022), 13041–13051.

[54] Renode. [n. d.]. Renode/Renode: Renode - Antmicro's open source simulation and Virtual Development Framework for complex embedded systems. https://github.com/renode/renode

[55] Emily Ruppel, Milijana Surbatovich, Harsh Desai, et al. 2022. An Architectural Charge Management Interface for Energy-Harvesting Systems. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 318–335. https://doi.org/10.1109/MICRO56248.2022.00034

[56] Adrian Sampson, Andre Baixo, Benjamin Ransford, et al. 2015. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. In *University of Washington Technical Report*. https://dada.cs.washington.edu/research/tr/2015/01/UW-CSE-15-01-01.pdf.

[57] Nicholas D Sanders, Richard M Everson, Jonathan E Fieldsend, and Alma AM Rahat. 2019. Bayesian search for robust optima. *arXiv preprint arXiv:1904.11416*.

[58] Seun Sangodoyin, Frank T Werner, Baki B Yilmaz, Chia-Lin Cheng, Elvan M Ugurlu, Nader Sehatbakhsh, Milos Prvulović, and Alenka Zajic. 2020. Side-channel propagation measurements and modeling for hardware security in iot devices. *IEEE Transactions on Antennas and Propagation* 69, 6 (2020), 3470–3484.

[59] Sivert T Sliper, Domenico Balsamo, Nikos Nikoleris, William Wang, Alex S Weddell, and Geoff V Merrett. 2019. Efficient state retention through paged memory management for reactive transient computing. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[60] Sivert T. Sliper, William Wang, Nikos Nikoleris, Alex S. Weddell, and Geoff V. Merrett. 2020. Fused: Closed-Loop Performance and Energy Simulation of Embedded Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 263–272. https://doi.org/10.1109/ISPASS48437.2020.00046

[61] Ankur Verma, Ayush Goyal, Soundar Kumara, and Thomas Kurfess. 2021. Edge-cloud computing performance benchmarking for IoT based machinery vibration monitoring. *Manufacturing Letters* 27 (2021), 39–41.

[62] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. 2012. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, https://datatracker.ietf.org/doc/html/rfc6550. Accessed: 2024-11-07.

[63] Chih-Hsuan Yen, Hashan Roshantha Mendis, Tei-Wei Kuo, et al. 2023. Keep in Balance: Runtime-reconfigurable Intermittent Deep Inference. *ACM Trans. Embed. Comput. Syst.* 22, 5s, Article 124 (Sept. 2023), 25 pages. https://doi.org/10.1145/3607918

[64] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 41–53.

A LLM Prompts Used in This Study

This appendix provides the detailed LLM prompts used in our research study, divided into two main conversations: the "Approximation Engine" and the "Makefile Generator."

A.1 Context Aware App Analyzer

Listing 6: System prompt.

```
You are "CheckMate," an LLM tool used for applying code
     approximations (approximate computing techniques).
     Our goal is to reduce program clock cycles to reduce
      energy consumption. For this, we are willing to
     accept some output errors.
```

Listing 7: Generate a summary of the code base.

```
Here is the codebase of the application:

{complete_code_base}

Create a small summary of the app and its purpose. Then,
     for each function, write a brief summary of its
     purpose.
```

Listing 8: Detailed function discription.

```
Let's discuss {function_name} function.

What is the purpose of {function_name} function?
What are its inputs and outputs? How does it accomplish
     its purpose?
How would its output interact with the rest of the code?
```

Listing 9: Function selection prompt.

```
Now, identify which functions should be approximated. For
      all functions in the code, please reply in this
     format:

# Function Name

Resoning for approximating or not approximating function.

# List of functions
{
  "function_1": "approximate",
  "function_2": "do not approximate",
  ...
  (all functions in code)
}
```

A.2 Approximation Engine

Listing 10: System prompt.

```
You are "CheckMate", an LLM tool designed to apply code
     approximations-techniques that intentionally
     introduce minor computational errors to reduce clock
      cycles, leveraging tolerance for inaccuracy in
     certain applications. The applications you are
     dealing with relate to batteryless IoT devices,
     where reducing power consumption and optimizing for
     low-energy environments is critical.

We will follow this flow to apply approximations:

    1. Planning/Annotation Step: After determining the
         function's purpose, you will assess whether it is
          safe to approximate. If so, you will annotate
         the code, adding comments where approximations
         can be applied, along with descriptions of what
         and how to apply them.

    2. Approximation Step: You will then be prompted
         again to apply the approximations you previously
         annotated.

These steps will be repeated for each function in our
     program's codebase. If a function contains calls to
     other functions, the conversation history of applied
      approximations for the called functions will be
     provided, enabling you to apply more effective
     approximations.

{code_base_summary}
```

Listing 11: Approximation identification.

```
Now, let's move to the planning step. For the given
     function, {function_name}, what are the possible
     approximations that can be performed?

If there exists a knob variable (a variable that, when
     changed, increases or decreases the level of
     approximations), state that it is a candidate.

A point to note, this code is meant for {
     platform_architecture} architecture, so your
     suggested changes should be possible within the
     architecture.
```

Listing 12: Approximation implementation.

```
Now, for the {function_name} function, approximate the
     code using the approximations highlighted in the
     previous prompt. The goal is to modify this function
      to reduce the number of clock cycles it uses. Focus
      on the identified areas of potential improvement
     and apply the described approximations. Additionally,
      for any of the knob variables identified or created,
      ensure they are declared at the top of the function
      within the /* Knob Variables Declaration Start */
     and /* Knob Variables Declaration End */ comments.

Return the following:

   The approximated code.
   A list of knob variables.
   A list of ranges for each knob variable.
   The step size by which to increment each knob
        variable; two options: Real or Integer.
```

Listing 13: Approximation JSON conversion.

```
{add_error}
Reformat the code you just generated into a JSON object.
     The JSON should contain the following information:

   1. approximated_code: The full block of code that you
        generated.
   2. knob_variables: A list of all the variable names
        that can be tuned or adjusted in the code. These
        are the "knobs."
   3. knob_ranges: The possible ranges of each knob
        variable (in the form of minimum and maximum
        values or specific values, if applicable).
   4. knob_increments: The increments by which each knob
        variable can change. Specify whether the
        increment is a Real number or an Integer.

Format the output as a clean and structured JSON object
     as described here:

{output_instuctions}
```

Listing 14: Few-shot prompt to guide LLM efficient loop perforation.

```
Here is an examples of loop perforation. If applying loop
      perforation only apply in this format. No other
     format of loop perforation is ever acceptable. Do
     not add in the iteration step, only the condition
     step.

#### Original un-perforated function

```c
void susan_edges(in, r, mid, bp, max_no, x_size, y_size)
   uchar *in,
   *bp, *mid;
int *r, max_no, x_size, y_size;
{
  /*Knob Variables Declaration Start*/
  int loop_skip = 2;
  float precision_scale = 0.9;
  /*Knob Variables Declaration End*/

  float z;
  int do_symmetry, i, j, m, n, a, b, x, y, w;
  uchar c, *p, *cp;

  memset(r, 0, x_size * y_size * sizeof(int));

  for (i = 3; i < y_size - 3; i += loop_skip) /*
       @Approximation applied [No.1] [Loop Perforation] */

    for (j = 3; j < x_size - 3; j += loop_skip) /*
         @Approximation applied [No.1] [Loop Perforation]
         */
    {
      n = 100;
      p = in + (i - 3) * x_size + j - 1;
      cp = bp + in[i * x_size + j];
    }
}
```

Perforated function

```c
void susan_edges(in, r, mid, bp, max_no, x_size, y_size)
   uchar *in,
   *bp, *mid;
int *r, max_no, x_size, y_size;
{
  /*Knob Variables Declaration Start*/
  int loop_perforation_factor = 0.2;
  float precision_scale = 0.9;
  /*Knob Variables Declaration End*/

  float z;
  int do_symmetry, i, j, m, n, a, b, x, y, w;
  uchar c, *p, *cp;

  memset(r, 0, x_size * y_size * sizeof(int));

  int loop_truc1 = (y_size - 3) * loop_perforation_factor
        /* truncating the loop */
  int loop_truc2 = (y_size - 3) * loop_perforation_factor
        /* truncating the loop */
  for (i = 3; i < loop_truc1; i++) /* @Approximation
       applied [No.1] [Loop Perforation] */
    for (j = 3; j < loop_truc2; j++) /* @Approximation
         applied [No.1] [Loop Perforation] */
    {
      n = 100;
      p = in + (i - 3) * x_size + j - 1 - n;
```
```

```
      cp = bp + in[i * x_size + p];
    }

  return cp;
}
```

Listing 15: Precision scaling example to help LLM understant safe and effective way to precision scale.

```
Here is an examples of precision scaling.
### Original unscaled function

```c
void calculate_image_gradient(int *gradient, float *image
     , int width, int height)
{
    int i, j;
    float gx, gy, magnitude;

    for (i = 1; i < height - 1; i++) /* Calculate
         gradients for all pixels */
        for (j = 1; j < width - 1; j++)
        {
            gx = (image[(i - 1) * width + (j + 1)] - image
                 [(i - 1) * width + (j - 1)]) +
                2 * (image[i * width + (j + 1)] - image[i *
                     width + (j - 1)]) +
                (image[(i + 1) * width + (j + 1)] - image
                     [(i + 1) * width + (j - 1)]);
            gy = (image[(i - 1) * width + (j - 1)] + 2 *
                 image[(i - 1) * width + j] + image[(i - 1)
                  * width + (j + 1)]) -
                 (image[(i + 1) * width + (j - 1)] + 2 *
                     image[(i + 1) * width + j] + image[(i
                     + 1) * width + (j + 1)]);
            magnitude = sqrt(gx * gx + gy * gy);
            gradient[i * width + j] = (int)magnitude;
        }
}
```

Scaled function with **Precision Scaling**

```c
void calculate_image_gradient(int *gradient, int *image,
     int width, int height)
{
    int i, j;
    int gx, gy, magnitude;

    for (i = 1; i < height - 1; i++) /* Calculate
         gradients for all pixels */
        for (j = 1; j < width - 1; j++)
        {
            gx = (image[(i - 1) * width + (j + 1)] - image
                 [(i - 1) * width + (j - 1)]) +
                2 * (image[i * width + (j + 1)] - image[i *
                     width + (j - 1)]) +
                (image[(i + 1) * width + (j + 1)] - image
                     [(i + 1) * width + (j - 1)]);
            gy = (image[(i - 1) * width + (j - 1)] + 2 *
                 image[(i - 1) * width + j] + image[(i - 1)
                  * width + (j + 1)]) -
                 (image[(i + 1) * width + (j - 1)] + 2 *
                     image[(i + 1) * width + j] + image[(i
                     + 1) * width + (j + 1)]);
            magnitude = abs(gx) + abs(gy); /* Approximation
                 by avoiding sqrt */
            gradient[i * width + j] = magnitude; /* Direct
                 assignment as int */
        }
}
```

A.3 Makefile Generator

Listing 16: System prompt.

```
You are an AI assistant that generates Makefiles based on
     a given list of source files. A Makefile is a file
    used by the make utility to manage the build
    automation of projects. The goal is to compile the
    source code files into an executable or library.

Instructions:

   1. You will be provided with a list of source files.
   2. Generate a Makefile that compiles these files into
        an executable named main.
   3. Include the necessary rules to compile object
       files from the source files.
   4. Create a clean rule to remove all object files and
        the executable.
   5. Use gcc as the compiler.
```

Listing 17: Makefile generation

```
Given a list of files in a directory, output a Makefile
    to compile the application.

   files = {files_list}

   Output only the Makefile content, no other text. Your
        exact output will be pasted into the Makefile (
       so do not include "```"). The command that will
       be run is just "make main". Do not use the -
       Werror or -Wall flag."
```
```