Towards Just-In-Time Software Approximations

Lucas Reis[1], **Lucas Wanner**[1], **Sandro Rigo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas, SP – Brazil

{lucas.reis,lucas,sandro}@ic.unicamp.br

Abstract. *Software-level approximations, such as loop perforation, function replacement, and memoization, can significantly enhance application performance and energy efficiency during compile time. However, approximating compilers often require extensive user intervention and lack the capability for real-time adaptation. This paper presents RAAS, a framework that integrates just-in-time recompilation with an automated evaluation system to create a general-purpose software approximation system with minimal user involvement. Our framework can apply input-aware approximations without needing a separate testing phase by continuously monitoring the target application and recompiling code blocks. We evaluated the framework with a set of resilient benchmarks while also comparing its performance with a similar framework focused on static compilation of approximations. Our findings demonstrate speedups of up to 6.3x with quality degradation limited to 30%, achieving competitive results to a static compilation with a shorter convergence time.*

Resumo. *Aproximações em nível de software, como a perforação de laços, substituição de funções e memoização podem melhorar o desempenho e consumo energético de uma aplicação de forma significativa em tempo de compilação. Entretanto, aproximadores a nível de compilação muitas vezes exigem intensa intervenção do usuário e pecam na capacidade de adaptação em tempo real. Esse trabalho apresenta RAAS, um framework que integra recompilação just-in-time com um sistema de avaliação automática para criar um ambiente de aproximação em nível de software de propósito geral que requer intervenção mínima do usuário. Nosso framework pode aplicar aproximações adaptáveis a diferentes entradas sem necessitar de uma fase de testes ao monitorar continuiamente o desempenho da aplicação alvo e recompilando trechos do código. Avaliamos o framework com um conjunto de benchmarks resilientes a aproximação, também comparando seu desempenho com um framework similar focado aproximações por compilação estática. Nossos testes demonstram speedups de até* 6.3*x com redução de qualidade limitado a 30%, alcançando resultados competitivos à compilação estática com reduzido tempo de convergência.*

1. Introduction

Approximate computing can be applied at different levels, from circuit and hardware design to applications and runtime systems, to improve performance by sacrificing accuracy in results. The goal of approximations is to accept degradation in output quality in exchange for disproportional gains in performance. At the software level, gains are typically achieved as reductions in execution time for critical parts of applications, resulting in lower energy consumption and negatively impacting output quality in the process.

Although potentially powerful, approximate computing is typically limited to fault-tolerant applications. Most techniques explored in the literature aim to solve problems that are too specific [Sidiroglou-Douskos et al. 2011, Hoffmann et al. 2011, Rinard 2013] or require much knowledge from the programmer. Given that application resilience is not universal, state-of-the-art often suggests compromises to avoid approximations in forbidden regions, which would cause errors or crashes. These approaches expect the developer to dive into the entrails of an application to look for ways to annotate approximable code [Sampson et al. 2011, Sampson et al. 2015] or force the programmer to work with approximation in mind from scratch, developing more than one version of the functions that would be approximated [Baek and Chilimbi 2010, Ansel et al. 2009].

Resilience to approximations is also often input-dependent, such that the same application has variable approximation resistance for different sets of inputs. Runtime adaptation is proposed as a viable alternative to avoid these problems [Baek and Chilimbi 2010, Kemp et al. 2021, Gadioli et al. 2019, Pervaiz et al. 2022]. Although state-of-the-art solutions can provide automatic approximation, runtime evaluation, and minimal user intervention, none solve all three problems simultaneously.

This paper proposes the Runtime Adaptative Approximation System (RAAS), a framework capable of applying automatic approximations to general-use applications while adjusting the relaxation aggressiveness at runtime and requiring minimum effort from the user. The system intends to be completely online, recompiling sections of the application as needed while respecting quality constraints and evaluating gains in execution time. The contributions of this work are:

- An automatic runtime system that makes approximation decisions based on quality constraints and speedup.
- Approximation techniques applied in the context of Just-In-Time (JIT) compilation as the adaptation system.
- Evaluation of benchmark kernels using function replacement and loop perforation.
- Comparison with a general-purpose static-compilation framework

Our evaluation shows that, for a fixed maximum error rate, the approximations yield speedups of up to 6.3x with no more than 30% total error on output while attaining better convergence rates than ACCEPT, a general-purpose approximating compiler that relies on static analysis.

2. Related Work

The literature explores software-level approximations in different ways. This section contextualizes state-of-the-art research on compile-time and runtime approximations, discussing the impacts and differences of our work.

The foundation for software-level approximations starts with a set of publications that allow for compilation and runtime adjustments based on different constraints and necessities [Ansel et al. 2009, Liu et al. 1994, Lachenmann et al. 2007, Hoffmann et al. 2011]. Inspired by previous approaches, Green [Baek and Chilimbi 2010] is presented as a strictly online approximation tool that monitors and adjusts approximations at runtime based on output quality and performance with user-required approximate versions of functions and loops. Unlike our proposal, all previous works demand significant commitment from the user, requiring manual implementations of approximate versions of code regions instead of automatically detecting and applying approximations.

ACCEPT [Sampson et al. 2015] is proposed as a general-purpose framework that generates approximations at compile time and, given inputs and Quality-of-Service (QoS) metrics, combines different configurations and evaluates the generated binaries to find the Pareto-optimal curve of the QoS/speedup trade-off. ACCEPT excels in the broadness of target applications but relies on an intricate annotation system that requires application parsing for fine-grained approximation control. [Reis and Wanner 2021] increments ACCEPT with two new approximation techniques and experiments with the runtime system without user-defined annotation. The lack of runtime-adjustable approximations diminishes the impact of the no-annotations approach as the convergence time explodes. ApproxTuner [Sharif et al. 2021] is a three-part framework that provides a vast kit of approximations for CNNs on CPU and GPU kernels, providing an output quality system that predicts results before executing the application. It is a complete framework with an extensive surface to attack for approximations, but it still requires compile-time effort per application.

GOAL [Pervaiz et al. 2022] is a robust adaptation framework that targets Swift applications and provides runtime adaptations based on user-defined knobs but requires manual in-depth parsing of source code to apply annotations. mARGOt [Gadioli et al. 2019] proposes an autotuning framework that adjusts software knobs at runtime to meet end-user-specified application goals. The framework can automatically evaluate and adjust the desired knobs with minimal overhead using look-up tables in response to input variations and environment changes. MIPAC [Kemp et al. 2021] is a design framework with a lightweight quality controller based on iterative algorithms. The framework can predict the number of iterations required to deliver optimal performance given an acceptable quality degradation. Both MIPAC and mARGOt differ from our approach by failing to provide automatic detection of approximation opportunities.

Our research aims to establish a unified framework that overcomes the limitations of existing methods, delivering significant performance improvements in general-purpose applications. Our framework offers a comprehensive solution by integrating the strengths of previous approaches, including automatic detection of opportunities, fine-grained runtime-adjustable approximations, and minimal user effort. Inspired by ACCEPT and its compiler-level approximation approach, we provide a similar array of techniques and a standardized approach for quality evaluation. We further solidify the validity of this approach by comparing both frameworks in section 4.4.

3. The Runtime Adaptative Approximation System

We propose the Runtime Adaptative Approximation System (RAAS) to provide dynamic approximations at runtime. This framework includes a dynamic compiler that evaluates application output and makes compilation decisions by adjusting approximation levels to target lower execution times.

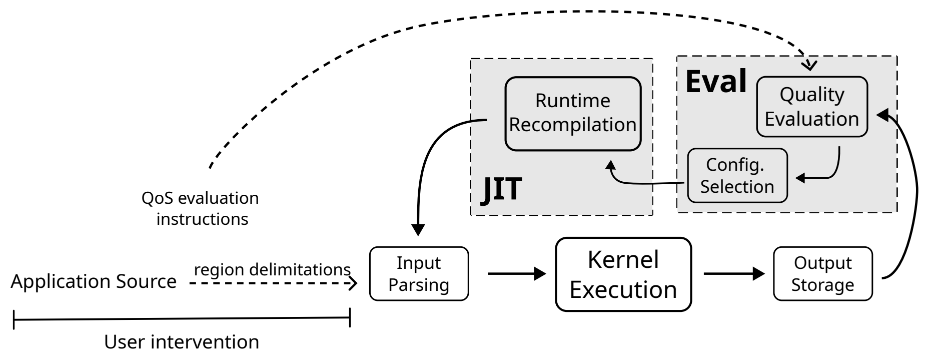


Figure 1. Overview of the runtime approximation framework

The framework is divided into two parts: an evaluation system and a JIT compiler. The evaluation system monitors the execution time and output of the application to compose the error and speedup rates for the computation-intensive application sections. Based on these metrics, a heuristic decision-making subsystem proposes a new set of approximations, called a configuration, to be provided to the application. Fig. 1 provides an overview of the framework and its components.

3.1. Approximation Techniques

The framework uses a modular design, where approximation techniques are LLVM passes that offer functionalities for the evaluation and recompilation systems. These passes provide an analysis segment that can parse functions to detect approximable code regions, which the framework uses before application execution to build its list of possible approximations. The approximation phase is composed of the modifications applied to LLVM Intermediate Representation (IR), compiled and executed by the JIT when a configuration is selected.

The framework's modular structure makes adding new approximation techniques straightforward. New approximations can be added as long as they provide analysis for approximable regions and transformations for approximations with varying levels of aggressiveness.

For this work in progress, we provide two techniques, the first being functional approximation, which replaces function calls with faster and less precise counterparts. As a proof of concept, the FastApprox [Poya 2017] library was used to replace mathematical functions, such as logarithmic and exponential computations, with a set of up to four variations of the same function with varying degrees of approximation. The second technique is loop perforation, widely used in the literature [Sidiroglou-Douskos et al. 2011]. It consists of skipping loop iterations at a variable rate to reduce computation times.

3.2. Target Applications and Limitations

RAAS adapts at runtime to any application within the context of its proposed approximation techniques as long as it regularly updates its output during execution. We present the tool as capable of taking control back from the application in timely frequencies, reevaluating and reapplying approximations given changes on input or the execution environment. These changes are measured as impacts on output quality and/or execution time variations. Adaptations can only be made when new changes occur, so the framework requires updates on output results to take context back. Thus, applications must provide constant output during execution, as the frequency at which the JIT reevaluates approximations is the same as when the application delivers new outputs. This can be seen in Fig. 1, where evaluation is always followed by output storing.

Following the previous requirements, we propose the tool targeting applications that run constantly and are prone to environmental changes, such as variable inputs and/or workload demands. High-performance computing (HPC) and server applications are key targets of this work, similar to previous proposals for approximations at runtime in the literature [Baek and Chilimbi 2010].

It is important to note that runtime compilation introduces overheads compared to compile-time approximation. Furthermore, the evaluation subsystem also adds some overhead since quality evaluation is somewhat expensive and may be a limitation for some applications, as discussed in Section 4. The framework applies aggressive approximations to reduce the impact of such overheads while monitoring for increased error metrics or crashes.

3.3. User Intervention

As RAAS is proposed to require minimal intervention from the end user, it is designed to reduce annotation constraints and significantly limit application intervention. For each target application, the user-required interventions are twofold: a quality metric and source code region delimitations to signal framework context changes.

Quality metric requirements are plentiful in the literature [Gadioli et al. 2019, Reis and Wanner 2021, Parasyris et al. 2021, Mitra et al. 2017, Pervaiz et al. 2022, Kemp et al. 2021] to allow automatic tools the ability to evaluate the quality of the approximate output in contrast to their precise counterparts. As quality metrics are application-dependent, the framework presented in this document requires a similar intervention. The tool asks the user for a simple script that presents functions to denote how to store application results in memory for precise and approximate iterations and a function that compares these results and returns an error rate.

Given the nature of automatic runtime approximation adjustments, the framework also requires users to apply two source code modifications: to mark sections of code to be considered regions of interest (RoI) and specify when to give context back to the runtime system for quality evaluation. This is done by simple function calls handled by the compiler to trigger context changes. RoIs allow the framework to measure execution time without input/output overhead interference and signal new output updates, which enable context changes back to the evaluation system to decide on a new configuration. Focusing the execution time on RoIs also discourages the framework from making changes to sections of code that we would not wish to approximate, such as input processing.

This solution differs from previous systems, requiring significantly less source code intervention. Our approach only requires signalizing when applications start and end computation phases and when the output phase stops. Decisions for approximations are made automatically and without user intervention. Conventional annotations, on the other hand, require signalizing which regions are forbidden from approximations within kernel phases of applications. This effort involves application source code parsing, fine-grained modifications, and, thus, more in-depth knowledge of the program that is being targeted, often resulting in extensive trial and error. Note that, as seen in Fig. 1, user intervention stops before the execution of the framework. After the application starts running, all decisions are automated. A concrete example of the interventions can be seen in section 4.1.

3.4. Search space and evaluation

The number of single opportunities largely depends on the application size for general-purpose approximations. The key advantage of software-level approximation is combining different configurations for multiple approximations to achieve better speedups while minimizing errors [Reis and Wanner 2021]. However, ensuring a configuration set is acceptable regarding speedup and error rates is only possible at runtime. Thus, one of the advantages of runtime approximations stems from the possibility of evaluating valid configurations on the fly. Still, targeting large applications often proves impossible to test all configurations, given that search spaces are exponential. Therefore, our approach for the framework is a modular structure capable of accepting different heuristics for approximations, which provides a way for the runtime system to analyze the current state of affairs and make decisions for the subsequent configurations to approximate (or not).

As a proof-of-concept, we present a simple heuristic that takes a greedy approach to approximation. We start by testing each opportunity alone and scoring based on error rates and speedup. Where S is the speedup in the last iteration and E is the output error, the score is calculated as follows: $score = (1 - 1/S)/E$. Then, we sort the list by score and apply approximations greedily as long as the current opportunity improves speedup without surpassing the error limit.

3.5. Crash avoidance

Section 3.2 notes that runtime approximation brings overhead from the compilation and evaluation systems, which more aggressive approximations must compensate for. Moreover, some approximations may provide more critical problems than high error rates, such as application crashes. Given that this is a work in progress and dealing with runtime crashes is a difficult task, the framework currently relies on manually informing limitations for opportunities that would yield fatal crashes, which adds effort on the user to monitor application execution and signal forbidden approximations.

4. Evaluation and Results

This section presents our experimental pipeline to ensure the framework's validity and shows each application's speedup and error rates.

4.1. Execution environment and benchmarks

To evaluate the efficacy of the framework, a set of benchmarks known to be resilient to approximations was selected from the PARSEC and Mibench benchmark suites [Bienia 2011, Guthaus et al. 2001]: blackscholes, bodytrack, FFT, ferret, fluidanimate, and swaptions. Quality metrics were based on absolute difference and distance to precise (non-approximated) results, normalized to fit the error range required by the framework based on the weighted average percentage error (WAPE) metric:

```
...
nRuns =  atoi(argv[4]);
// Read input and setup data structures
...
for (int k = 0; k < nRuns; ++k) {
  JIT_roi_begin();
  // computing kernel
  bs_thread();
  JIT_roi_end();
  // store output
  JIT_evaluate_output();
}
...
```

Figure 2. Changes to the blackscholes application.

$$E = \frac{\sum_{i=1}^{n} |p_i - a_i|}{\sum_{i=1}^{n} p_i}, \tag{1}$$

where p_i is a precise value, and a_i is an approximate value for all the values provided on the output.

As presented in section 3.2, applications must follow a pattern of constant output storage to be a target of our framework. Therefore, all benchmarks have been adapted to follow this structure to ensure viability inside the tool. The key modifications include instrumenting the application's core with a repeated loop for a defined number of iterations. As such, each iteration represents an entire execution of the original benchmark, which the framework interprets as a significant interval to measure output quality and reevaluate approximations. The code snippet in Fig. 2 provides a simplified version of the *blackscholes* benchmark main function. The modifications to adapt the benchmark to our evaluation are marked in red. In contrast, the blue lines denote the interventions the framework requires to act on the approximations, as seen in Section 3.3. We emphasize that user intervention stops at the main function, which requires no further application knowledge for the user to annotate specific code regions (such as functions) inside the application kernel. A similar approach is used for all benchmarks tested.

Given that each benchmark is executed repeatedly from the same input, each iteration is equal and suffers no variability except for the changes in approximation. Although this is not equal to a real context in which input and execution environments vary between iterations, we argue that this fix is sufficient for a proof of concept of the capabilities of runtime approximations and is an approach for validation similar to previous work in the literature [Baek and Chilimbi 2010].

All benchmarks are compiled to LLVM IR and fed to LLVM's default optimizer using the `-O3` flag for optimizations before being fed to the framework. Further interventions are done on blackscholes, in which we manually mark a key function to forbid inlining to prevent it from inlining to the main function. This solves a native problem from runtime frameworks: we cannot recompile the main function, as it is only called once. Therefore, any code inside it is incapable of approximations.

The executions were also compared with a statically compiled version of the same benchmark without intervention from the framework. The static binary is compiled in

Table 1. Testing environment overview for each benchmark

Benchmark	Input Size	Approx. Space	Iterations to Converge	Total Iterations
blackscholes	large	5	183	1000
swaptions	large	45	1425	3000
ferret	medium	45	1152	2000
fluidanimate	medium	75	4171	5000
fft	large	9	51	500
bodytrack	medium	123	3830	5000

Table 2. Summary of the results.

	Error		Speedup (x)			Overhead	
			Region of Interest		Max.	Abs (s)	Rel (%)
Benchmark	Training	Convergence	Training	Convergence			
blackscholes	<0.1	0	7825	7929	6.3	11.2	8.4
swaptions	100	25	4	4	2.3	204.2	4.6
ferret	100	29.7	4.2	3.9	2.8	-1.6	<0.01
fluidanimate	3.1	1.8	115	114	4.9	-50.3	-2.6
fft	92	3.2	1.6	1.1	1.2	-0.66	-0.3
bodytrack	2.9	0.3	1.6	1.6	1.3	91.9	6.3

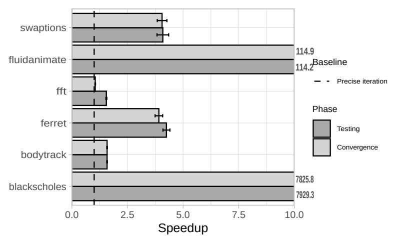
Clang using a standard compilation pipeline, including compiling any benchmark with the `-O3` optimization flag. All experiments use LLVM 16.0.6; every result except output errors is displayed as the average of 10 executions.

Table 1 provides more details on the evaluation pipeline for each benchmark. The approximation space is the number of base opportunities for approximation for each benchmark, considering that a single opportunity approximates a region of code for a specific technique. As noted in Section 3.4, the framework will combine opportunities following a simple greedy approximation approach while monitoring execution times and error rates for each iteration. The total iterations were manually selected during experimentation based on the time it takes for each benchmark to converge for a final solution for the selected input. After convergence, the evaluation process is still executed on each iteration to simulate a possible change in environment and inputs, as output quality analysis is a considerable overhead in executing each iteration.

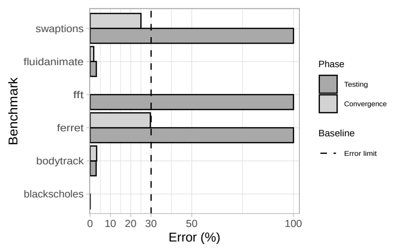
4.2. Results

Table 2 summarizes the results for each benchmark. Note that speedups are measured in two contexts: *region of interest* (RoI) and direct comparisons with precise executions from statically compiled binaries with Clang. The reason for this distinction is structured in the way the framework evaluates gains in speedups. Measurement of regions of interest is how the JIT evaluates changes in execution times (as per section 3.3), providing useful data to understand decision-making within the framework. At the same time, measuring speedup when compared with a static execution is how we justify the framework's usability as a general approximation tool, given that approximations must be impactful both inside the framework and in global execution time measurements.

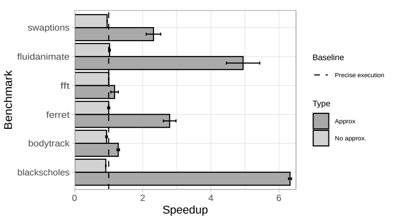
RoI speedups and error rates are also divided between testing and convergence phases to better envision the decision-making done by the framework when faced with high error rates and the impact they cause on execution time changes. Raw overhead is measured in the last column using a simple methodology: the framework is executed



(a) Region of Interest speedup



(b) Iteration error



(c) General speedup

Figure 3. Overall results for speedup and error

for the benchmark with function replacement disabled. This ensures that the entire evaluation/recompilation pipeline occurs, but no approximations are applied to the selected functions. We compare the execution time for this approach with those from precise executions, and any difference is interpreted as framework overhead.

Fig. 3 presents the table in a visual form to better visualize the results, with the standard deviation for each bar.

4.3. Discussion

Comparing error rates and RoI speedups is vital in understanding the difference in resilience to approximation for each benchmark and the decisions made by the framework to mitigate these differences. While benchmarks such as blackscholes and bodytrack never attain significant errors and can be approximated easily, swaptions, ferret, and FFT receive approximations that create errors way above the fixed upper limit, requiring the evaluation system to discard such relaxations. Despite these drawbacks, RAAS was capable of achieving valid gains for most of these benchmarks after adaptation, except for FFT.

The discrepancy between output error between applications posits a variance of resilience between benchmarks, which is justified by the results of blackscholes and fluidanimate, as both applications achieve significant speedups without ever exceeding the upper error limit. Following a different trend, bodytrack is the only application that lacks significant speedups without surpassing our upper error limit, indicating small resilience to the proposed techniques. Further analysis with new techniques better suited for similar applications may show different trends.

A similar discrepancy exists between the speedups observed in regions of interest and those in the overall application. Our data indicate that this gap is partly due to overhead introduced by the framework, which is not captured in RoI measurements. However, the primary cause of this discrepancy lies in the difference between the execution times of RoIs and the full application. In some benchmarks, such as blackscholes, this effect is particularly pronounced since the application devotes a substantial portion of its runtime to input/output processing, especially for larger datasets. Since our approximations do not target these regions, RoI-based speedups can sometimes present a misleading picture of overall performance.

Lastly, we address the impact of overhead. Table 2 highlights a noticeable variation in overhead across benchmarks, which we attribute to differences in evaluation metrics. For instance, benchmarks like FFT and swaptions produce small outputs. In contrast, benchmarks like blackscholes and fluidanimate require more extensive output processing due to a large number of values to analyze, which scale with input size. Benchmarks with minimal overhead indicate that the runtime compilation infrastructure does not hinder the use of approximations, shifting the burden instead to the evaluation system. We argue that even in applications with considerable overhead, the trade-offs provided by approximations remain advantageous, outweighing the potential drawbacks.

4.4. Comparisons with ACCEPT

To reinforce the framework's capabilities, we compare it against ACCEPT, a static and general-purpose approximation framework. The decision to choose ACCEPT stems from its similarity in the methodology of approximations with a runtime evaluation subsystem that compares output results based on a custom error metric and is targeted to sustain upper error limits. The framework is capable of the same approximations implemented in our proof-of-concept, as provided by [Reis and Wanner 2021], but with a static compilation approach.

Table 3. Comparison between RAAS and ACCEPT. For speedups and approximation space, higher is better. For the rest, lower is better.

		Speedup (x)			
Benchmark	**Convergence Time (x)**	**Region of Interest**	**Max.**	**Error (x)**	**Approx. Space (x)**
blackscholes	1.03	49.65	1.31	0	0.83
swaptions	0.49	1.57	0.88	7.8	0.47
fluidanimate	0.27	0.15	0.43	300113	1.10
fft	1.04	1.1	1.2	0.7	1.5

Table 3 compares the results of running benchmarks from our original tests using the ACCEPT evaluation system, with ACCEPT's values serving as a baseline. Two benchmarks were excluded due to a bug in the LLVM version (3.2) used by ACCEPT, which prevented their compilation. Benchmarks were also tested without the runtime modifications for constant output delivery described in Section 4.1.

As expected, RAAS speedups generally followed a similar pattern, with a few exceptions. The most prominent case was fluidanimate, which achieved significantly better RoI performance with ACCEPT due to its reliance on a static compilation system, unlike RAAS's JIT-based approach. ACCEPT was able to leverage a highly effective approximation in fluidanimate's main function, leading to notable speedups and lower error rates. In contrast, RAAS could not approximate the main function, as it is called only once and thus cannot benefit from runtime recompilation. While adapting fluidanimate to fit RAAS's framework could address this limitation, we have deliberately avoided such modifications to ensure a fair comparison with minimal user intervention across both frameworks.

Excluding fluidanimate, RAAS provided larger RoI speedups for all benchmarks, even in applications with a smaller approximation space. Despite that, these gains are not always translated into proportional maximum speedups, which we attribute to the inherent overheads of runtime approximations. Despite these drawbacks, we believe that the results present an optimistic view on runtime approximations, especially considering the impact on reducing convergence time and the possibility of adaptations on the fly for target applications.

5. Conclusions

Software-level approximation has been explored by various frameworks to provide trade-offs between error rates and execution times in general-purpose applications. This paper introduces RAAS, a framework that supports runtime adaptation across diverse applications with minimal user input, offering automatic evaluation and compilation using general-purpose approximation techniques.

With minimal source code changes, the system applies various approximation combinations while monitoring execution times and output accuracy, aiming to maximize speedups while keeping errors within acceptable limits. Our proof-of-concept evaluation, using two approximation techniques across six benchmarks, achieved significant speedups while maintaining error rates within acceptable bounds.

To enhance the framework's longevity, we propose improvements to the evaluation system such as adaptive heuristics to account for environmental changes, reducing overhead by using larger intervals between output evaluations. Future work includes adding a crash-handling subsystem to eliminate user intervention and expanding techniques like automatic parallelization and precision scaling.

We believe that ensuring user-friendly approximation paves the way for the future of software-level approximate computing, and we argue that our framework is a step in the right direction.

6. Acknowledgments

This work was supported in part by QuintoAndar, the National Council for Scientific and Technological Development (CNPq) grants 402467/2021-3 and 405940/2022-0, Coordination for the Improvement of Higher Education Personnel (CAPES) grant 88887.954253/2024-00 and Finance Code 001, and Unicamp FAEPEX grant 2477/23.

References

Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). Petabricks: A language and compiler for algorithmic choice. In *PLDI*.

Baek, W. and Chilimbi, T. M. (2010). Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

Gadioli, D., Vitali, E., Palermo, G., and Silvano, C. (2019). mARGOt: A dynamic autotuning framework for self-aware approximate computing. *IEEE Transactions on Computers*, 68:713–728.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., and Brown, R. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *IEEE WWC*.

Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. (2011). Dynamic knobs for responsive power-aware computing. *SIGPLAN Not.*, 46(3):199–212.

Kemp, T., Yao, Y., and Kim, Y. (2021). Mipac: Dynamic input-aware accuracy control for dynamic auto-tuning of iterative approximate computing. In *ASP-DAC*.

Lachenmann, A., Marrón, P. J., Minder, D., and Rothermel, K. (2007). Meeting lifetime goals with energy levels. *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*.

Liu, J., Shih, W.-K., Lin, K.-J., Bettati, R., and Chung, J.-Y. (1994). Imprecise computations. *Proceedings of the IEEE*.

Mitra, S., Gupta, M. K., Misailovic, S., and Bagchi, S. (2017). Phase-aware optimization in approximate computing. In *CGO*.

Parasyris, K., Georgakoudis, G., Menon, H., Diffenderfer, J., Laguna, I., Osei-Kuffuor, D., and Schordan, M. (2021). HPAC: evaluating approximate computing techniques on hpc openmp applications. In *SC*.

Pervaiz, A., Yang, Y. H., Duracz, A., Bartha, F., Sai, R., Imes, C., Cartwright, R., Palem, K., Lu, S., and Hoffmann, H. (2022). Goal: Supporting general and dynamic adaptation in computing systems. In *Onward*.

Poya, R. (2017). Fast approx library. `https://github.com/romeric/fastapprox`.

Reis, L. and Wanner, L. (2021). Functional approximation and approximate parallelization with the accept compiler. In *SBAC-PAD*.

Rinard, M. (2013). Parallel synchronization-free approximate data structure construction. In *USENIX Wksp. on Hot Topics in Parallelism*.

Sampson, A., Baixo, A., Ransford, B., Moreau, T., Yip, J., Ceze, L., and Oskin, M. (2015). Accept: A programmer-guided compiler framework for practical approximate computing. *U. Washington UW-CSE-15-01*.

Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. (2011). Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6).

Sharif, H., Zhao, Y., Kotsifakou, M., Kothari, A., Schreiber, B., Wang, E., Sarita, Y., Zhao, N., Joshi, K., Adve, V. S., Misailovic, S., and Adve, S. (2021). Approxtuner: a compiler and runtime system for adaptive approximations. In *PPoPP '21*, page 262–277.

Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *ACM ESEC/FSE*, page 124–134.