Multithread Approximation: A new OpenMP construct

João B. Oliveira[1], Rogério A. Gonçalves[1], João Fabrício Filho[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87.301-899 – Campo Mourão – PR – Brazil

joliveira.2022@alunos.utfpr.edu.br,{rogerioag,joaof}@utfpr.edu.br

Abstract. *This study presents a new construct in* `OpenMP` *designed to facilitate the implementation of approximate computing techniques within parallel programming environments. By integrating approximation methods such as task dropping, loop perforation, and floating-point relaxation, the proposed construct aims to enhance performance and energy efficiency while maintaining acceptable accuracy levels. Experimental results on benchmark applications demonstrate a trade-off of up to* 490.83%*, with* 55.1% *of quality loss, highlighting the potential and limitations of approximate computing in parallel contexts.*

1. Introduction

Approximate Computing (AC) explores acceptable margins of error in application results in exchange for performance and energy efficiency [Mittal 2016]. Domains such as data analysis, pattern recognition, image processing, and signal processing in general require results that accept some degree of imprecision [Kugler 2015]. This imprecision is determined by the application context and its perception limitations, involving quality metrics and noise present in the data [Xu et al. 2016].

AC techniques relax precision across computing layers from circuits to algorithms [Que et al. 2023]. Techniques that explore approximation in hardware, although general-purpose, require modifications that add cost to system design. Software modifications offer benefits with less implementation cost. In the software layer, modifications that can be generalized from different types of computations include approximation techniques in the compiler [Reis and Wanner 2021].

Approximations through function memoization [Tziantzioulis et al. 2018], loop perforation [Li et al. 2018a], or floating-point relaxation [Schkufza et al. 2014] are some compiler-implemented techniques that achieve performance and energy efficiency. However, implementing these techniques individually requires significant changes to the application's source code. Conversely, techniques employing pragmas and programming interfaces, like `OpenMP` [omp], streamline the process of incorporating these methods into applications.

This work explores compiler approximation techniques for performance gains through code annotations using the `OpenMP` interface. Based on previous results [Oliveira et al. 2024], we implement constructs that allow the application of loop perforation and memoization with just the addition of pragmas to the code. In this way,

Thanks to the Araucária Foundation for the financial support and scholarship grant - Project 01947-SISPEQ/UTFPR.

the code requires minimal changes to use AC techniques while benefiting from the existing gains in the `OpenMP` interface, including code parallelization.

Our results demonstrate performance improvements of up to 490.83%, with only a 55.1% reduction in quality. On average, the quality of the applications is 62.63% compared to the original versions. These findings highlight the potential of using AC techniques in compilers, requiring minimal changes to the application source code while still achieving significant performance gains.

In the following sections, we provide further details on the techniques used and the experiments conducted. Section 2 discusses the specific approximation techniques applied in the compiler. Section 3 reviews related works in the field of approximation computing (AC) within parallel programming. Section 4 introduces our proposed `OpenMP` constructs. Section 5 outlines the methodology employed in our experiments. Section 6 analyzes the experimental results. Section 7 offers core insights into the use and implementation of these constructs. Finally, Section 8 presents conclusions and future directions for optimizing approximation computing in `OpenMP` environments.

2. Approximation Techniques into Compiler

Task Dropping is a technique where a portion of the computational work is skipped. It involves specifying a region and a parameter that represents the dropping rate. During the scheduling of the task, the runtime dynamically adjusts the workloads, deciding which portions should be executed and which can be dropped.

Loop Perforation, similar to task dropping, is a technique that skips some workloads to gain performance or energy efficiency. Loops are among the most frequently executed parts of code, making them prime targets for optimizations [Bacon et al. 1994]. The basic implementation of this algorithm involves modifying the loop's induction variable so that some iterations are simply not executed [Hoffmann et al.].

Floating-point Relaxation numbers involve an approximation in their representation due to the impossibility of representing continuous, infinite numbers with a finite number of bits. Consequently, floating-point representations can introduce errors during arithmetic operations due to rounding and truncation when the result cannot be represented within the limited precision of the floating-point format [Monniaux 2008]. To mitigate some imprecision introduced during these operations, compilers often avoid applying certain optimizations that could otherwise enhance software performance. However, some compilers, such as `Clang` and `GCC`, allow the user to enable these optimizations, even if it could lead to inaccuracies, through the flag `--fast-math`, which activates aggressive optimizations by compilation unit [gcc , cla]. Additional, `MSVC` supports the use of the `pragma float_control`, enabling these optimizations within a specified region of code [msv].

Temporal Memoization is an approximate memoization technique that leverages the temporal order of a function's invocation to optimize its performance. Traditionally, memoization focus on caching results based on inputs, ensuring that repeated inputs result in instant output retrieval [Michie 1968]. However, a distinctive feature of memoization with temporal locality is that it primarily focuses on the output. This strategy hinges on the

observation that many calculations performed within a specific context and temporal locality are either identical or closely related. To harness this potential, these computations are stashed in a cache, available for quick access to improve overall performance. Moreover, this technique allows for various levels of granularity in caching, such as global, per call site, or context-aware strategies. Not all functions are suitable for this memoization context. Only functions that return a scalar type, lack side effects, and maintain consistency with these principles within the same function can be effectively incorporated into the memoization process. [Tziantzioulis et al. 2018]

3. Related Work

Lashgar [Lashgar et al. 2018] brings the concept of loop perforation to `OpenACC`, which is a standard that works similarly as `OpenMP`. Based on some annotations in the code, the compiler generates code that applies the dropping of iterations.

Sculptor [Li et al. 2018b] proposed compiler optimizations and a runtime that could perform the loop perforation, based on the dynamic instructions and the iterations of a loop. SampleMine [Jiang et al. 2022] is a framework that implements the loop perforation techniques to gain performance on subgraph data mining.

Vassiliadis [Vassiliadis et al. 2014], propose an approximated task approach to gain energy efficiency, in which some operation where some tasks were substitutes by approximated ones or simple not executed at all.

ApproxHadoop [Goiri et al. 2015], distributed systems can also benefit from this paradigm by using the idea of task dropping, where part of the computations is simply not executed. Rinard [Rinard 2006] has also studied part of this idea of dropping tasks and brings some discussions about when and why this technique can be applied.

HPCA [Parasyris et al. 2021] is a framework that uses different approximation techniques to provide an easy way to apply this paradigm to the code. Like `OpenMP` and `OpenACC`, it works by using annotations in the code and providing a runtime that applies the techniques of memoization or loop perforation.

In contrast with these other works, our work builds upon the established foundation of `OpenMP` and leverages its parallel programming model to integrate approximate computing techniques. This not only simplifies the use of approximation within parallel computing but also opens doors for wider adoption and exploration of this field.

4. Proposal

The current proposal introduces a new construct in `OpenMP` designed to facilitate the implementation of approximate algorithms. Similar to other `OpenMP` constructs, this implementation relies on `pragma` annotations to identify regions of code that can be approximated. Integrating these annotations with the existing framework ensures that the approximation techniques are portable and easily adaptable to different code regions without disrupting existing codebases.

The `pragma approx` 1 annotation identifies a code region for approximation and specifies the technique to be applied within that region. More than one clause can be used within the construct, allowing for the use of combined clauses. However, the current

implementation does not support more than one approximated algorithm at a time within the specified clauses.

Código 1. Syntax of the `approx` constructor.

```
1 #pragma omp approx [clause[[,] clause]]
```

Floating point relaxation, defined as `fastmath` 2, is a clause that introduce a mechanism for applying relaxation and optimizations, as specified in the `fast-math` flag of the `Clang` compiler [cla], to designated regions of code.

Código 2. Syntax of the `fastmath` clause.

```
1 #pragma omp approx fastmath
```

Loop perforation, defined as the `perfo` 3 clause, is instrumental in implementing the loop perforation algorithm [Hoffmann et al.]. This clause needs to be used in combination with the `for` clause. A perforation modifier specify the type of perforation, offering choices such as `init`, `fini`, `small` and `large`, along with a value to specify the range of iterations that can be discarded.

Código 3. Syntax of the `perfo` clause.

```
1 #pragma omp approx for perfo(modifier,drop-rate)
```

Task dropping, defined as the `drop` 4 clause, implements the algorithm for dropping some iterations. This clause needs to be used in combination with `taskloop`. As in the loop perforation, this clause is going to drop some iterations of the loop, based on the drop rate passed as a parameter to `drop`.

Código 4. Syntax of the `drop` clause.

```
1 #pragma omp approx taskloop drop(drop-rate)
```

Temporal memoization, defined as the `memo` 5 clause, implements the algorithm for memoization. The clause needs to be used in conjunction with the `shared` clause, so that the variables address can be saved to its results be later verified, other important aspect of this clause is the use of the `threshold` modifier that if not specified will assume that all values can be memoized without verifying the output loss.

Código 5. Syntax of the `memo` clause.

```
1 #pragma omp approx memo shared([variables])
2                          [threshold(error-limit)]
```

5. Methodology

The experiments were made in the Intel® Core™ i7-4790 CPU @ 3.60GHz with 4 physical cores and support for 8 threads, 32GB of RAM. All the tests were made in the Ubuntu 22.04.4 LTS.

Most of the applications used for the benchmark come from the *Rodinia* benchmark suite, which represents a wide variety of computationally and memory-intensive tasks. The applications chosen cover different problem domains, including finance, fluid

dynamics, clustering, and molecular dynamics. A brief description of the ones chosen can be found on Table 1.

Benchmarks	Definition
Blackscholes [Yazdanbakhsh et al. 2017]	A mathematical model that estimates the price of options by solving a partial differential equation
CFD [Che et al. 2009]	An unstructured grid solver for 3D Euler equations in compressible flow
K-means [Che et al. 2009]	A clustering algorithm that divides data into K sub-clusters based on features
Particle Filter [Che et al. 2009]	A statistical estimator that tracks a target's location using noisy measurements and predicted paths
LavaMD [Che et al. 2009]	Calculates particle potential and relocation due to mutual forces in 3D space

Table 1. List of applications used.

For most applications, the new annotations were used alongside existing `OpenMP` annotations to manage parallelism effectively. In cases where multiple annotations were required, they were applied strategically to different sections of the code, typically targeting the main computational function where the majority of the workload occurs. Both the baseline (non-approximated) and the approximated versions of the applications were compiled using the -O3 flag, which enables aggressive optimizations such as function inlining, loop unrolling, and vectorization. This ensures that both versions are fully optimized for performance, allowing for a fair comparison of the impact of approximation techniques.

To compare the accuracy loss of the applications, two metrics were used, the Mean Absolute Percentage Error (MAPE) 1, and the Miss-Classification-Rate (MCR) 2. Both used the n as the number of elements in the data array, A_t is the t-th element in the accurate array and F_t is the t-th element in the approximate array. In the MCR, the ι was used as an operator that returns 1 when the condition is true.

The MAPE was selected to analyze the applications, because it provides a percentage-based measure of deviation from the accurate results, making it useful for understanding the relative error across different benchmarks. MCR was used specifically to evaluate the K-means application, as it's a metric that measures the rate of incorrect classifications.

$$MAPE(A, F) = \frac{1}{n}\sum_{t-1}^{n}|\frac{A_t - F_t}{A_t}| \tag{1}$$

$$MCR(A, F) = \frac{1}{n}\sum_{t-1}^{n}\iota(A_t \neq F_t) \tag{2}$$

The applications were tested with different entries of sizes that vary from a tiny to a large entry, and each one of the applications were tested with a different number of threads (1, 2, 4, 6, and 8). All applications were executed 10 times to minimize environmental variations. Random perforations in the *perfo* and *drop* clauses used a fixed seed for consistency. Speed-up was measured by comparing the number of cycles between the original and approximated versions of the applications.

6. Results

The evaluation focused on performance improvements and accuracy trade-offs when applying approximate computing techniques to the chosen benchmarks. Speedup measurements were based on the number of cycles in each application.

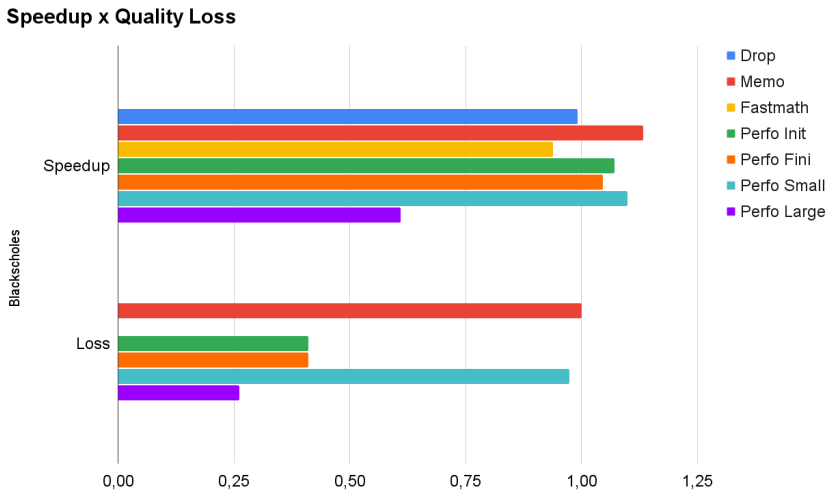

Figure 1. Speedup vs. Quality Loss of Blackscholes.

Blackscholes: Annotations were added to the `bs_thread` function, which already contained `pragma` annotations from the Rodinia benchmark suite. As shown in Figure 1, the technique `memo`, `perfo small`, `perfo init`, `perfo fini`, and achieved speedups of 13.35%, 10% 7.07% and 4.69%, respectively. In terms of output quality, `memo` and `perfo small` experienced significant degradation. Specifically, `memo` that resulted in a loss of 100%, and `perfo small` that had a 97.36% loss.

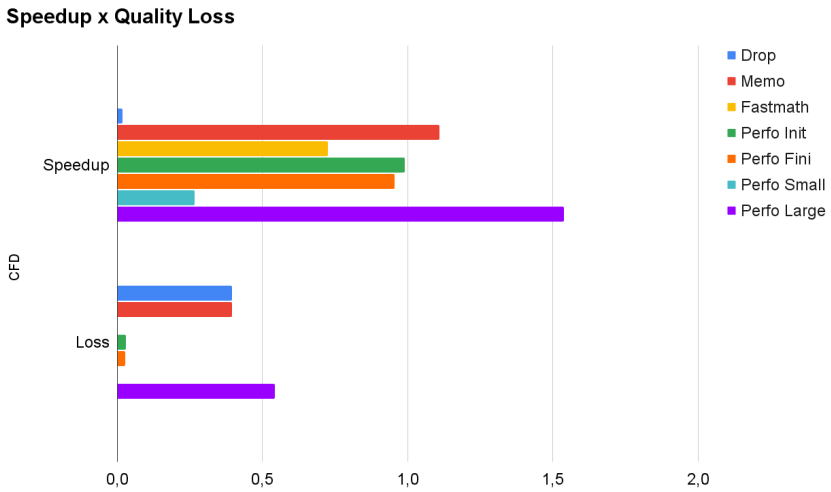

Figure 2. Speedup vs. Quality Loss of CFD.

CFD: In this application, several functions already had annotations, so only the `compute_flux` function was annotated with approximation techniques due to its high computational intensity. Figure 2 displays the speedup and accuracy loss associated with these annotations. The results show that `memo` achieved a speedup of 10.91%, while `perfo large` provided the most significant speedup at 53.19%. Other techniques resulted in some degradation in speedup. Unlike other applications, CFD generally supported approximation techniques with minimal quality loss. The highest quality loss was observed with `perfo large`, which had a loss of 54.26%.

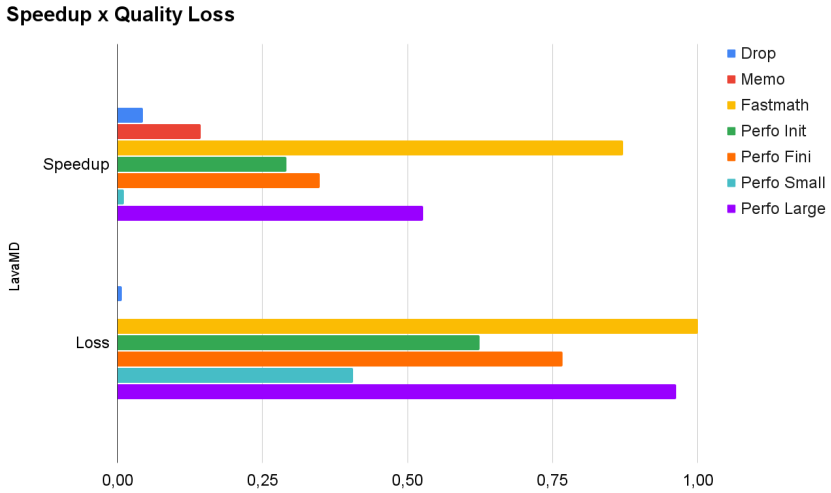

Figure 3. Speedup vs. Quality, Loss of LavaMD.

LavaMD: Annotations were applied to the `kernel_cpu` function in this program. As depicted in Figure 3, LavaMD showed the poorest performance among all the tested applications, with substantial speedup losses across all techniques. The quality

degradation was also among the worst, with `memo` and `perfo large` resulting in a complete loss of 100% in quality.

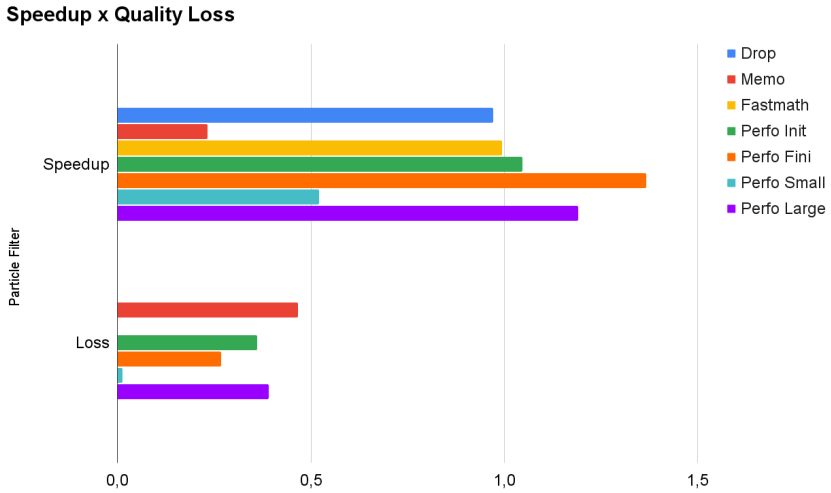

Figure 4. Speedup vs. Quality Loss of Particle Filter.

Particle Filter: Like the CFD application, this application also features multiple annotations, all within the `particleFilter` function. Except for the `fastmath` annotation, which does not alter the code structure, other annotations were applied solely to the loop handling the motion model and particle filter likelihood. The results are presented in Figure 4. In this application, `perfo fini` achieved a speedup of 36.62%, `perfo large` achieved 19.13% and `perfo init` achieved 4.74%. In terms of quality loss, none of the techniques resulted in more than 50% loss, with `memo` showing the highest loss at 46.65%.

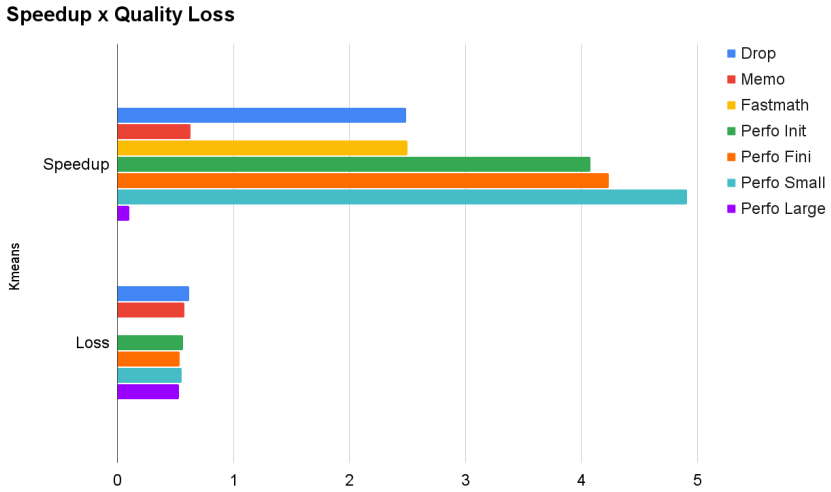

Figure 5. Speedup vs. Quality Loss of K-means.

K-means: Annotations were applied to the main loop of the

kmeans_clustering function in this program. Figure 5 presents the results for this application. Overall, it achieved the best performance among the tested applications, with a speedup gain of up to 4 times the original application. Specifically, the perfo small technique delivered a remarkable speedup of 490.83%, while drop achieved 248.65%, fastmath 249.95%, perfo init 407.67%, and perfo fini 423.31%. Despite these gains, the quality loss was significant, averaging 50%, with the highest degradation observed with drop, which resulted in a quality loss of 61.62%.

The results also highlight the limitations of using approximation techniques in parallel programming, primarily due to issues with quality and execution overhead. Without runtime result checks, approximation techniques often lead to suboptimal quality outcomes. Additionally, the overhead from OpenMP calls in these techniques can be substantial, especially in iterations where approximation is unnecessary.

The applications tested exhibited some variability in results, but an overall analysis of the techniques is as follows:

- fastmath: This clause introduced minimal friction, showing neither notable speedup nor quality loss.
- drop: This clause also failed to deliver significant speedup and did not achieve a balanced quality loss.
- memo: This clause caused the most substantial quality loss in the final results.
- perfo: Results varied across different implementations, but overall, the perfo large version provided the best speedup, though with poor quality outputs.

The results varied significantly between applications. Among the tested applications, kmeans emerged as the most effective for using these techniques, achieving the best balance between speedup and quality.

7. Core Insights

As a final insight, techniques such as loop perforation and memoization have demonstrated significant performance improvements, although this often comes with a trade-off in result accuracy. Applications already employing some form of approximation, like K-means and Particle Filter, show better outcomes in both performance and speedup, indicating that while not all applications benefit from these techniques, those that do can achieve substantial gains. For example, K-means, which depends on distance calculations for cluster assignment, can see marked performance enhancements by skipping some of these calculations through loop perforation, without drastically compromising accuracy. Similarly, other applications that do not require strict sequential consistency can also gain from these approximation methods

Regarding OpenMP, the complexity of its codebase, driven by its role in parallel computing across various platforms, poses additional challenges. Much of the performance overhead observed arises from internal checks and computations necessary to ensure correct execution, contributing to the overall inefficiency. Part of the performance issues seen, particularly with the perfo technique, could be mitigated by using Clang's optimization passes to restructure loops instead of relying on runtime calls. Another way to reduce overhead is by implementing runtime checkers that monitor result quality. If a result falls below the expected quality threshold, the application could revert to its normal

execution flow. While this would introduce some initial overhead, performance would stabilize after a few executions, reducing overhead and improving output quality overall. This approach could help optimize both performance and accuracy without sacrificing too much on either front.

8. Conclusion

The main goal of this work was to integrate the paradigm of approximate computation into the `OpenMP` infrastructure, combining it with parallel computations to explore new optimization opportunities. This research contributes significantly by providing a detailed evaluation of various approximation techniques within the `OpenMP` framework, highlighting their potential benefits and limitations.

This study extends existing work by applying approximation techniques, such as loop perforation and memoization, in a parallel computing context using `OpenMP`. While previous research has explored these techniques individually or in different contexts, our work comprehensively analyzes their performance and quality trade-offs when combined with `OpenMP`. We implemented and evaluated five different approximation algorithms, uncovering significant limitations in their application within this framework. Key issues included quality degradation and execution overhead, primarily due to the overhead associated with calling runtime functions and the absence of runtime result checks to improve output quality and reduce overhead.

As future work, we plan to optimize the implementation further and introduce a runtime checker to verify application results. The runtime checker will support addressing the current limitations by enhancing output quality and mitigating execution overhead.

References

[cla] Clang Compiler User's Manual — Clang 18.0.0git documentation. https://clang.llvm.org/docs/UsersManual.html#cmdoption-ffast-math.

[msv] float control pragma. https://learn.microsoft.com/en-us/cpp/preprocessor/float-control?view=msvc-170.

[omp] Llvm/openmp 19.0.0git documentation. https://openmp.llvm.org/.

[gcc] Optimize Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-ffast-math.

[Bacon et al. 1994] Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420.

[Che et al. 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54.

[Goiri et al. 2015] Goiri, I., Bianchini, R., Nagarakatte, S., and Nguyen, T. D. (2015). ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–397, Istanbul Turkey. ACM.

[Hoffmann et al.] Hoffmann, H., Misailovic, S., Sidiroglou, S., Rinard, M., and Agarwal, A. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures.

[Jiang et al. 2022] Jiang, P., Wei, Y., Su, J., Wang, R., and Wu, B. (2022). SampleMine: A Framework for Applying Random Sampling to Subgraph Pattern Mining through Loop Perforation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 185–197, Chicago Illinois. ACM.

[Kugler 2015] Kugler, L. (2015). Is "good enough" computing good enough? *Communications of the ACM*, 58(5):12–14.

[Lashgar et al. 2018] Lashgar, A., Atoofian, E., and Baniasadi, A. (2018). Loop Perforation in OpenACC. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 163–170, Melbourne, Australia. IEEE.

[Li et al. 2018a] Li, S., Park, S., and Mahlke, S. (2018a). Sculptor: Flexible approximation with selective dynamic loop perforation. In *Proceedings of the International Conference on Supercomputing*, volume 11, pages 341–351. ACM.

[Li et al. 2018b] Li, S., Park, S., and Mahlke, S. (2018b). Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 341–351, Beijing China. ACM.

[Michie 1968] Michie, D. (1968). "memo" functions and machine learning. *Nature*, 218(5136):19–22.

[Mittal 2016] Mittal, S. (2016). A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, 48(4):1–33.

[Monniaux 2008] Monniaux, D. (2008). The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3):1–41.

[Oliveira et al. 2024] Oliveira, J., Gonçalves, R., and Fabrício Filho, J. (2024). OpenMP em Direção à Aproximação: Loop Perforation e Multithreading. In *Anais da XV Escola Regional de Alto Desempenho de São Paulo*, pages 49–52, Porto Alegre, RS, Brasil. SBC.

[Parasyris et al. 2021] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, St. Louis Missouri. ACM.

[Que et al. 2023] Que, H.-H., Jin, Y., Wang, T., Liu, M.-K., Yang, X.-H., and Qiao, F. (2023). A Survey of Approximate Computing: From Arithmetic Units Design to High-Level Applications. *Journal of Computer Science and Technology*, 38(2):251–272.

[Reis and Wanner 2021] Reis, L. and Wanner, L. (2021). Functional approximation and approximate parallelization with the accept compiler. In *2021 IEEE 33rd Inter-*

national Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pages 188–197.

[Rinard 2006] Rinard, M. (2006). Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334, Cairns Queensland Australia. ACM.

[Schkufza et al. 2014] Schkufza, E., Sharma, R., and Aiken, A. (2014). Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Notices*, volume 49, pages 53–64, New York, New York, USA. ACM Press.

[Tziantzioulis et al. 2018] Tziantzioulis, G., Hardavellas, N., and Campanoni, S. (2018). Temporal Approximate Function Memoization. *IEEE Micro*, 38(4):60–70.

[Vassiliadis et al. 2014] Vassiliadis, V., Parasyris, K., Chalios, C., Antonopoulos, C. D., Lalis, S., Bellas, N., Vandierendonck, H., and Nikolopoulos, D. S. (2014). A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing. arXiv:1412.5150 [cs].

[Xu et al. 2016] Xu, Q., Mytkowicz, T., and Kim, N. S. (2016). Approximate Computing: A Survey. *IEEE Design and Test*, 33(1):8–22.

[Yazdanbakhsh et al. 2017] Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., and Lotfi-Kamran, P. (2017). AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test*, 34(2):60–68.