Approximate Computing: Greedy and Selective Loop Perforation

Yuhang Ning, Chen Huang, Yuanzhe Zheng, Houming Chen, and Yucheng Huo
University of Michigan Ann Arbor, MI
{dlning, kouchin, yuanzhez, houmingc, yhuo}@umich.edu

***Abstract*—Loop perforation is an established technique within the domain of approximate computing that strategically modifies loop execution to enhance performance at the cost of reduced accuracy. This project examines two variants of loop perforation: the traditional greedy loop perforation as introduced by [1], and the load-based selective loop perforation, an enhancement suggested by [2]. We implemented both methods and evaluated their impact on computational efficiency and accuracy on four benchmarks that we designed. Our evaluation results indicate that while both methods accelerate the program execution with an acceptable loss in accuracy, the load-based selective loop perforation indeed outperforms the traditional greedy method by achieving a better trade-off between speed and accuracy in most scenarios, especially when the program has complex structures.**

I. Introduction

In numerous computation tasks, exact precision is not always required, and results that are approximately correct may suffice. For instance, Monte Carlo methods provide approximate solutions through stochastic simulations [3], while machine learning algorithms generate probabilistic models that approximate complex phenomena—capturing essential features without demanding absolute precision [4].

This concept of relinquishing some accuracy for gains in performance or energy efficiency is called approximate computing, a practice increasingly prevalent across diverse areas of computer science. Many previous studies ([1], [5]–[7]) have shown that employing certain software techniques in approximate computing can result in significant performance enhancements while only marginally compromising accuracy, a trade-off that is often preferred in various applications.

A widely recognized technique within approximate computing is task skipping, where certain computation tasks are deliberately omitted to reduce execution time or energy consumption, albeit at the cost of some output quality [8]. One such technique is loop perforation, where the loops are modified to skip certain instructions within certain iterations [1].

The initial loop perforation method was introduced in [1], referred to as the greedy loop perforation. This technique transforms the program to skip iterations periodically. This technique of loop perforation has proven to be a simple, general, and effective method of task skipping.

However, a subsequent study [2] points out that the greedy loop perforation has a significant limitation - it only skips iterations completely with all internal instructions and periodically with a fixed rate, which may not always be the optimal approach. To address this, [2] introduced two improved techniques: selective loop perforation and dynamic loop perforation. In selective perforation, loops are modified to skip a flexible subset of their instructions. This method is further categorized into three types: instruction-level, load-based, and store-based selective loop perforation. Each type employs a different strategy to select the instructions to skip. In dynamic loop perforation, loops are modified to skip a flexible subset of their iterations. [2] then integrated the selective and dynamic loop perforation methods into a composite method known as selective dynamic loop perforation [2]. This composite method transforms loops to skip a dynamic subset of their instructions within a dynamic subset of their iterations [2]. The evaluation conducted by [2] demonstrated that this approach could indeed achieve greater speedups with lower error budgets compared to the traditional greedy methods.

In this course project, we aim to reproduce and extend some of these findings. We implemented the original greedy loop perforation proposed in [1] and, due to the complexity of the full selective dynamic loop perforation method, we implemented the load-based selective loop perforation proposed in [2]. We then assessed the performance of these two techniques across four small benchmarks that we designed to ascertain that the load-based selective loop perforation approach indeed offers a better trade-off between speed and accuracy than the traditional greedy loop perforation.

Our contribution can be summarized as follows:

1) We implemented the greedy loop perforation proposed in [1] and the load-based selective loop perforation proposed in [2]. Then, we compared the performance of these two perforation methods on four benchmarks that we designed.
2) Our experimental findings suggest that while there are instances of comparable performance between the two approaches, the load-based selective loop perforation generally offers more speedups with lower error budgets. This is particularly evident in programs that have complicated structures. These results support that the load-based selective loop perforation introduced in [2] is indeed a successful enhancement to the greedy loop perforation technique originally proposed in [1].

II. Related Work

In this section, we provide a more detailed description of the existing works that we are implementing in this project.

A. Greedy Loop Perforation

Sidiroglou et al. [1] propose a greedy exploration algorithm to search the loop perforation space of a program and generate a set of loops and their corresponding perforation rates that meet a specified accuracy bound. This approach uses a heuristic scoring metric to prioritize which loops to perforate and at what rates, aiming to maximize performance while adhering to the accuracy constraint.

The process starts with an application, a set of representative inputs, an accuracy bound, and a set of tunable loops generated by a previous algorithm. The algorithm then computes a subset of these loops that have the highest priority based on a scoring function. This function calculates scores based on the harmonic mean of the estimated performance increase and accuracy loss for perforating a given loop. Loops are chosen in a sequentially prioritized manner, starting with the one that is predicted to offer the best trade-off between speedup and accuracy loss, always checking that the resulting accuracy after perforation is above the specified bound and that the speedup is greater than 1.

The greedy algorithm explores the loop perforation space by selecting only one perforation rate for each loop, specifically the rate with the highest priority as determined by the scoring function. This method contrasts with exhaustive exploration by not considering all possible combinations of loops and rates, making it more efficient in terms of computational resources and time required to find a satisfactory set of perforations. However, this efficiency might come at the cost of potentially missing more optimal combinations that could have been discovered with an exhaustive search, due to the greedy method's focus on immediate priority choices rather than exploring the wider space of possibilities.

B. Selective Loop Perforation

Li et al. [2] propose two adaptive loop perforation strategies in Sculptor, and one of them is called selective loop perforation. It is a technique where loops are transformed to skip a flexible subset of their instructions, unlike traditional perforation that skips a constant subset of iterations. This method does not necessitate skipping all instructions in the iterations. Instead, it allows for skipping a subset of instructions deemed unimportant to the outcome's accuracy, while continuing to execute instructions important for maintaining accuracy. This approach provides a more refined control over which instructions are executed and which are skipped, aiming to optimize the performance-accuracy trade-off by selectively targeting instructions for skipping within the loop's iterations.

The Sculptor paper categorizes the selective loop perforation into three types: *instruction-level, load-based, and store-based*. Each of them consists of three main stages: *selection stage, expansion stage, and transformation stage*. The selection stage picks the initial instructions that are qualified to be perforated. The expansion stage further picks other instructions that are closely related to the selected instructions. And the transformation stage finally converts the LLVM IR into a version with those instructions removed.

III. Implementation

A. Greedy Loop Perforation

The greedy loop perforation method, originally introduced by Sidiroglou et al. [1], serves as a fundamental approach in approximate computing to enhance performance or energy efficiency by skipping iterations of loops during execution. In its classic form, this method skips iterations in a fixed pattern, reducing the total number of loop iterations executed and thus, the computation time. While this technique straightforwardly reduces resource usage, its simplicity comes with the limitation of a uniform approach to perforation, which may not always yield optimal results in terms of maintaining a balance between accuracy and performance.

1) Selecting Target Loops: We selected innermost loops that are safe to perforate, focusing on those without critical dependencies or side effects that could lead to significant errors or failures in the program. To identify these loops, we implemented an LLVM Pass, LoopCountPass, to analyze the dependency graphs, and performed runtime checks in a Python driver program to ensure that perforating these loops would not disrupt essential program operations or data integrity.

2) Modifying Loops for Perforation: Once suitable loops were identified, our method employed a perforation technique that modifies the iteration mechanism of the loop. Specifically, the increment operation within the loop's control structure was altered to skip a predefined number of iterations, effectively reducing the number of times the loop body is executed. This alteration was performed at the level of the LLVM Intermediate Representation (IR) using the LLVM compiler infrastructure, with an LLVM Pass, LoopPerforationPass. The modification process involves the following steps:

- **Identification of Increment Instruction:** The loop's increment instruction, typically an addition operation that increases the loop counter, is identified.
- **Adjusting the Increment:** The identified addition operation is modified to increase the loop counter by more than one, depending on the specified perforation rate. This change directly reduces the number of loop iterations executed.

3) Automated Benchmarking and Evaluation with Python Driver Program: To automate the evaluation of our greedy loop perforation implementation, we developed a Python driver program that integrates with LLVM's compilation tools. This driver handles the compilation of C code into LLVM IR, applies the loop perforation transformations, and then compiles the modified IR back into executable form. It runs both the original and modified programs, capturing and comparing their outputs to assess the impact of perforation on performance and accuracy. The program measures execution times and computes the accuracy loss relative to a predefined error threshold, facilitating an empirical evaluation of the trade-offs involved.

The program also automates the selection of loops for perforation using LLVM's analysis passes to identify which loops can be safely modified without significant accuracy

loss. It tests various perforation rates to optimize the balance between execution speed and output quality. Command-line options allow the specification of file paths, function names, loop indices, and perforation rates, enhancing the flexibility and usability of the tool for a range of benchmarks and use cases. This streamlined approach allows for rigorous testing and fine-tuning of the loop perforation process, providing valuable insights into its practical implications in approximate computing.

B. Selective Loop Perforation

After examining the three types of selective loop perforation types from the Sculptor paper, we notice that the only difference among the three types lies in the selection stage. The instructions-level type selects all the instructions that don't make the program collapse and don't cause the error in the result to exceed the error budget. To use the error metric, a complicated run-time profiling pass is needed to keep track of the influence of each instruction during the run time. The load-based type just directly selects all the load instructions. The store-based type selects all the stores that don't cause the program to fail. To balance the difficulty and performance of the final project, we decide to combine the load and store-based strategies and not implement the instruction-level selection. Notice that even if some instructions may not be selected in the initial selection stage compared with the instruction-level strategy, those instructions are still likely to be chosen in the expansion stage. Therefore, we believe implementing the load and store-based strategies would still result in good performance.

1) Find Loop Candidates: All loops, including inner loops in a loop nest, are considered. We use the static profiling information to calculate the fraction of the frequency of all instructions inside the loop over the frequency of all instructions inside the program, and select the loops whose fractions are greater than a threshold set by us as loop candidates.

2) Selection Stage: Prior to picking the initial perforable instructions, we implement another pass, as shown in Fig.1.(a), to find all the instructions that are not perforable. Those instructions are: (a) Branches and their producers; (b) Instructions that output a pointer, with their producers; (c) All instructions related to the original loop induction variables. All of them are related to pointers and the control flow. Removing those instructions may break the program (e.g. segmentation fault, infinite loop), so they are all marked as not perforable. To correctly find all the producers of them, we implement a breadth first search pass to recursively find any predecessors of the instructions within the loop. Finally, inside the selection stage, as shown in Fig.1.(b), we select all loads and stores within the loop that are not marked as "not perforable", and mark them as "perforable".

3) Expansion Stage: Based on the selected instructions from the selection stage, the expansion stage further finds the instructions within the loop that: (a) Only use loop invariants and the values of the selected instructions, or (b) Their values are only used by the selected instructions. We also use a breadth first search algorithm to recursively find all those instructions and mark them as "perforable" as well. Throughout the search, any instructions that are marked as "not perforable" are skipped.

4) Transformation Stage: After the expansion stage, the set of instructions marked for skipping is expanded to include those that are tightly coupled with the initially selected instructions. In the transformation stage, the system needs to apply the necessary perforation strategy to the actual code, transforming it according to the selected and expanded set of instructions to be skipped. As discussed in [2], the intuitive solution just simply splits the basic block and inserts branch instruction before every target instruction. This causes too much control divergence therefore downgrades the program performance. To address this problem, we chose to implement one more sophisticated compiler optimization mentioned in [2], called loop unswitching optimization, which is designed to optimize loops with invariant conditional expressions. The compiler creates two versions of the loop body, one original and one perforated (with all target instructions remove). Then it inserts the branch instructions to loop preheader to select which version of loop to execute to skip perforated instructions, depending on the perforation rate. Compared to the brute force solution, this approach minimizes the number of redundant conditional branch instructions, reducing the runtime overhead of perforation.

IV. Evaluation

Table I contains the four benchmark and the corresponding error metric we used to analyze the error. Sum-to-n simply calculates the sum from 0 to n, and in our benchmark we used n being 10000000 (10 million). Nested involves random calculation including adding, multiplying, dividing a constant to output an integer using three layers of nested for loops. This is used to test our implementation's specially handling in nested for loops. Having output all being a single integer, those two benchmarks use error metrics of percentage and log scale percentage to find the extent of miscalculation.

Mat-mul, short standing of matrix multiplication, is a benchmark that performs matrix multiplication on floating point inputted matrices. 3-3 kernel is a benchmark that performs matrix convolution using a size 3x3 kernel. The output of the benchmark are all matrices, and Average Relative Error (ARE) metrics are used to evaluate the error rate:

$$\text{Average Relative Error} = \sum_{i=1}^{N} \left| \frac{y_i^* - y_i}{y_i} \right| \times \frac{1}{N} \qquad (1)$$

where y_i^* represents the tested result, and y_i represents the actual ground truth result.

This helps to measure the error rate of the perforated result as a percentage.

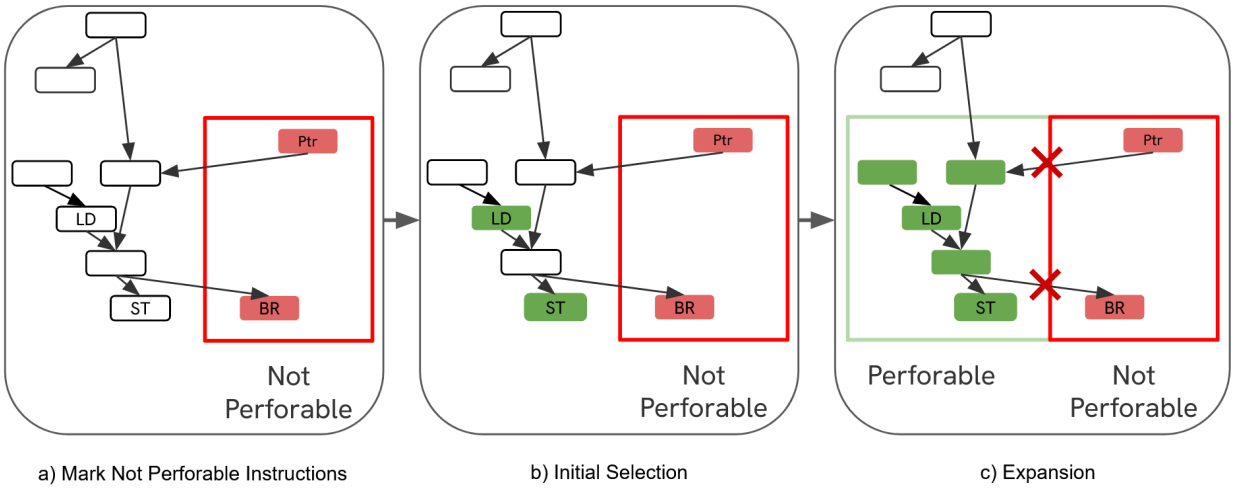


Fig. 1. Selection and Expansion Stages

TABLE I
TESTCASES AND ERROR METRICS

Testcase	Error Metric
sum-to-n	Percent, Log Scale Difference
nested	Percent, Log Scale Difference
mat-mul	ARE
3x3-kernel	ARE

V. RESULT AND ANALYSIS

According to Table IV, sum-to-n benchmark gives similar performance to the greedy implementation. This is because there are only a few instructions in each iteration. Selective perforation achieves a similar effect with greedy perforation if keeping the same perforation rate, since there are only limited instructions to be perforated and selective perforation's advantage of fine-grained perforation is not applicable. We toggled the perforated rate to let the error be as small as possible. As the minimum constant perforation rate is 2, the percent error of 50% is highly reasonable and the acceleration effect is similar. For the nested benchmark, the structures of loops become more complicated. There are a lot of nested loops where complex instructions exist in each loop (outer and inner). In this scenario, selective perforation clearly outperforms the greedy implementation by selectively perforating certain multiply/divide instructions within different loop scopes.

According to Table V, we still see similar performance between greedy and selective approach for mat-mul. This is because of the underlying nature behind the mat-mul kernel: instructions are highly dependent and short. For example, there is a "%2 = load memory" instruction in the innermost loop of mat-mul to load an element of the matrix into value %2. If we perforated this instruction, the actual part doing the multiplication will also be perforated as it depends on value %2. For the 3x3 convolution kernel, we see better ARE error and speed up as the instruction within iteration contains 9 pieces of multiplication and adding. The selective perforation pass can flexibly choose which of the instructions to perforate while keep the others. On the other hand, the greedy perforation method, applying a uniform skip pattern, does not distinguish between different types of operations and may inadvertently remove critical computations, leading to a significant decline in output accuracy without proportionate gains in performance. This highlights the advantage of selective perforation in scenarios where operations within the loops vary significantly in their impact on the final output and computational cost.

TABLE II
GREEDY: ERROR RATE AND SPEEDUP A

Testcase	Percent Error	Log Scale Difference	Speedup
sum-to-n	50.01%	0.30	20x
nested	61.18%	0.21	2x

TABLE III
GREEDY: ERROR RATE AND SPEEDUP B

Testcase	ARE Error	Speedup
mat-mul	24.89%	1.5x
3x3-kernel	35.10%	2.0x

TABLE IV
SELECTIVE: ERROR RATE AND SPEEDUP A

Testcase	Percent Error	Log Scale Difference	Speedup
sum-to-n	50.01%	0.30	19.8x
nested	37%	0.11	3.4x

TABLE V
SELECTIVE: ERROR RATE AND SPEEDUP B

Testcase	ARE Error	Speedup
mat-mul	24.70%	1.6x
3x3-kernel	28.30%	2.4x

VI. CONCLUSION AND FUTURE WORK

A. Conclusion

In our work, we explored the effectiveness of approximate computing, specifically though the topic of loop perforation techniques in compiler optimization. The proposed two loop perforation techniques were successfully implemented in LLVM, and we contracted the traditional greedy loop perforation in [1] with load, store-based selective loop perforation in [2]. Through the analysis across various benchmarks, we observed that selective loop perforation can generally achieve better trade-offs between performance and accuracy than its greedy counterpart. This is particularly evident in more complex structures, such as nested loops and operations involving multiple dependencies within each iteration. The ability to selectively target instructions for perforation allows for a more refined control over the computational resources, optimizing the program's performance without a substantial compromise on accuracy.

B. Future Work

In addition to our current findings, several possibilities remain open for further exploration:

1) Extended Benchmarks and Domains: Due to the time constraints, we did not test our perforations on more complex benchmarks. Future studies could expand the scope of benchmarks and explore additional domains where loop perforation might be beneficial.

2) Hybrid Approaches and Performance: Combining both greedy and selective strategies might yield even better optimization results. A hybrid approach could leverage the simplicity and quick execution of the greedy method for less critical sections of code and apply selective perforation where higher accuracy is required, or possibly run two perforation strategies simultaneously for better results.

3) Machine Learning Aided Loop Perforation: Implementing machine learning algorithms to predict the most beneficial perforation strategies could automate and further optimize the selection process. This could involve training models on a variety of code bases to learn patterns and predict optimal perforation points and rates.

4) Tool Integration and Usability: Developing user-friendly tools/interface that integrate these loop perforation techniques into standard development environments could facilitate broader adoption and experimentation.

APPENDIX A
SOURCE CODE

The source code of our greedy loop perforation algorithm can be found here: https://github.com/yuchenghuo/eecs583-final-project. The source code of our selective loop perforation pass and the testcases we use can be found here: https://github.com/kouchin-um/EECS583_LoopPerf.

REFERENCES

[1] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), p. 124–134, Association for Computing Machinery, 2011.

[2] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible approximation with selective dynamic loop perforation," in *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, (New York, NY, USA), p. 341–351, Association for Computing Machinery, 2018.

[3] M. H. Kalos and P. A. Whitlock, *Monte carlo methods*. John Wiley & Sons, 2009.

[4] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *International Static Analysis Symposium*, pp. 316–333, Springer, 2011.

[5] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 198–209, 2010.

[6] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 13–24, 2013.

[7] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 164–174, 2011.

[8] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 324–334, 2006.