PROCsimate: A Scheme for Approximating Procedures with Dynamic Quality Monitoring and Result Guarantees

Aurangzeb and Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
orangzeb@purdue.edu, eigenman@purdue.edu

***Abstract*—Approximating entire procedures in applications amenable to approximation can offer significant performance gains. This paper proposes PROCsimate, a function approximation scheme. PROCsimate provides efficient and fast function approximation in software, monitors quality of approximation, and offers guarantees about results. The scheme dynamically improves approximation over the course of application execution by capturing changing input behavior of the application and by speculatively training itself. The scheme improves the ease of use over existing schemes that require multiple trial-and-error runs for setting approximation parameters. Instead, the new method automatically selects these parameters and tunes them over time to produce acceptable results and honor guarantees. PROCsimate leverages idle cores in a system to offload some of the tasks from its critical path. It also introduces a multilevel hash table to dynamically enrich history which the scheme uses to build an approximate model of the candidate function.**

***Keywords*-approximate computing; function approximation; history-based piecewise approximation**

I. Introduction

Many applications in machine learning, image processing, computer vision, data analytics, simulations, gaming, audio, and video do not have strict demands on the exactness of results. They can tolerate a certain degree of inaccuracy in results. Approximate computing aims to tradeoff accuracy in these domains for increased performance and/or reduced power. Novel hardware [1], [2], [3], software [4], [5], [6], and hybrid [7], [8] techniques target approximation at different levels of granularity.

Approximating program procedures can offer significant performance benefits. Different contributions [9], [10], [11], [12], [13], [14], [15] have pursued function approximation. History-based piecewise approximation [16] is the latest scheme that has been shown to offer fast and efficient function approximation in software. This scheme employs polynomial approximation in multiple regions of function input, which the scheme forms according to the function invocation history. While the scheme performs well, it does not monitor quality and offer result guarantees. In order to get satisfactory results, the user may have to run the scheme multiple times with different parameters. Moreover, this tuning process would have to be repeated when the input to the application changes. The uniform version of the scheme, which divides the function input range in uniform-sized regions, provides fast approximation but has additional limitations. It requires profiling information to determine the input range of the candidate function. The uniform scheme stores the region information, training data, and coefficients of polynomials for each region in a hash table. The size of each region and number of regions (size of the hash table) are determined by the user-provided parameters. Once the scheme has created regions, they stay fixed. This prevents the scheme from dynamically improving the approximation by enriching training data and making changes into regions and their corresponding polynomials.

We introduce PROCsimate, a fast, efficient, self-improving, self-monitoring and easy-to-use function approximation scheme that offers result guarantees. It employs history-based uniform piecewise approximation as the underlying approximation strategy. Currently, the scheme uses polynomials in single variables offering significant performance improvements to a number of applications that have single parameter candidate functions, and it can be extended to support polynomials in multiple variables. PROCsimate does not require user-provided parameters except for the tolerable error bound needed to offer result guarantees. It automatically selects its parameters and tunes them over the course of the application execution to produce satisfactory results within the provided error bound. PROCsimate obviates the need for profiling information or multiple trial-and-error attempts and is thus easy to use. It can also handle applications whose inputs change across runs. The scheme harnesses idle cores to perform some of its tasks. It dynamically improves the approximation by capturing the changing input behavior during the application execution using dynamic online training and by speculative training. It introduces a custom hash table to support dynamic insertion and modification of regions, training data, and polynomials, helping the scheme dynamically improve approximation in a single run with scheme-selected parameters.

This paper makes the following contributions:

1) A novel procedure approximation scheme, PROCsi-

mate, that monitors quality, offers guarantees about results, and dynamically improves itself. The scheme is easy-to-use and does not require extra profiling phases or multiple trial-and-error attempts to find proper parameter settings for approximating an application efficiently.

2) A multilevel hash-table used by the scheme. This data structure may be of use for other applications.

The rest of the paper is organized as follows. Section II describes the underlying approximation strategy that PROCsimate employs. Section III describes the PROCsimate scheme including the mechanisms for dynamically improving itself, monitoring quality, and offering guarantees. The section also introduces a customized multilevel hash-table that allows PROCsimate to enrich the history dynamically and improve the approximations. Section IV discusses related work and Section V concludes the paper.

II. Background

The history-based piecewise approximation scheme [16] approximates functions by dividing their input range in uniform or non-uniform regions, based on invocation history, and applying lower-order polynomial approximation in those regions. During training, which happens online, the scheme stores the input-output pairs in the history, forms regions, and computes coefficients of polynomials for regions. During production, for each input, the scheme finds the appropriate region and evaluates the corresponding polynomial. The scheme comes in two flavors: the non-uniform scheme forms regions of different sizes; and the uniform scheme has equal-sized regions. The non-uniform scheme is realized with three different underlying storage and lookup mechanisms: an array (BSA); a binary search tree (BST); and a red-black tree (RBT). The uniform scheme is realized via a hash-table. The non-uniform scheme can provide better approximation than its uniform counterpart but is slower. The latter scheme requires an extra profiling run and does not allow dynamic expansion of history or formation of new regions.

The current history-based piecewise scheme [16] does not monitor quality and offer result guarantees. It does not improve approximation over time and finding proper parameter settings for an application requires multiple trial-and-error attempts. The next section provides novel mechanisms that overcome these limitations.

III. PROCsimate

This section introduces PROCsimate, an approximation scheme for procedures that is general, efficient, self-improving, self-monitoring, easy-to-use, and provides result guarantees. The scheme builds on history-based uniform piecewise approximation, which is a general and software-only scheme that has been shown to provide efficient and fast function approximation.

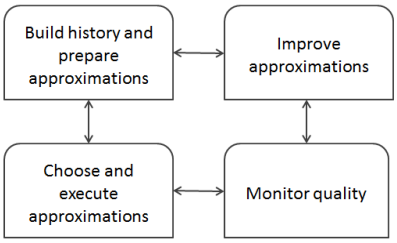

Figure 1. High-level tasks performed by PROCsimate.

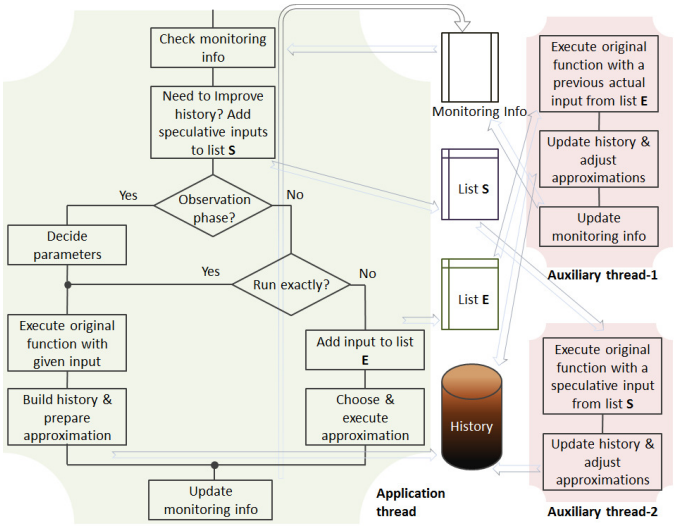

Figure 2. Specific tasks performed by application main thread and auxiliary threads under PROCsimate.

The following subsections describe the scheme in detail.

A. Overview of PROCsimate

Figure 1 describes the high-level tasks that PROCsimate performs. It builds history by storing training inputs and corresponding exact outputs and prepares approximations by forming regions, deciding polynomials for regions, and computing coefficients of polynomials. For production inputs, it chooses and executes approximations by finding the corresponding region and evaluating the associated polynomial. It monitors the output quality and improves approximations by updating history, forming new regions and polynomials, and adjusting existing regions and corresponding polynomials.

Figure 2 shows task details as they are performed by different threads under PROCsimate. The thread on the left is the application main thread, which executes the approximate function provided in replacement of the original function by PROCsimate. The threads on the right are the auxiliary threads that the scheme creates. These threads have lower priorities than the application thread and are expected to make use of idle CPU cores. For demonstration, the figure is showing two auxiliary threads, but in practice, the scheme will have more to harness all available cores. The inputs

for which the scheme invokes the original function in the application thread are called training inputs and those for which approximation is performed are called production inputs. In the middle column, Figure 2 shows the data that the scheme keeps. *History* stores inputs and corresponding exact outputs, region information, and polynomial information for each region. *Monitoring info* contains the bookkeeping and approximation quality information. It helps the scheme make certain decisions, described in Section III-D. *List **S*** contains a list of arbitrary inputs determined speculatively. Auxiliary threads (e.g. thread-2 in Figure 2) remove an input from this list, call the original function with the input, store it with the exact result in history, and update the approximation by adjusting regions and polynomials. *List **E*** comprises production inputs for which the scheme has recently performed an approximation in the application thread. Auxiliary threads (e.g. thread-1 in Figure 2) remove an item from the list, find the exact output by calling the original function for the removed input, compare it with the approximated output, and update the monitoring info, accordingly. They also update the history and adjust approximations.

The application thread consults the monitoring info and, if needed, determines the speculative inputs, which it adds to *List **S***. Speculative training is one of the ways by which the scheme improves its approximation. Improving approximations is further explained in Section III-B. The scheme goes through an observation phase in the beginning. In this phase, the scheme analyzes the input and tries to find suitable parameters. These parameters can later be adjusted by the scheme, as needed. In the observation phase, inputs are considered training inputs and the original function is called for them. These inputs and the exact function results are stored in the history. Based on the monitoring info, the scheme can also choose to treat an input in the regular (non-observation) phase as a training input. Otherwise, it will be considered a production input and approximation will be performed by locating the corresponding region and evaluating the associated polynomial against the production input. The production input is also stored in *List **E*** so that the auxiliary threads may find its exact result and compare against the approximation. Finally, the scheme updates the monitoring info in support of the decisions, described in Section III-D.

B. Improving Approximation

PROCsimate improves approximation over the course of the application execution. It is done using dynamic training, speculative training, and comparing approximation results against the exact results of production inputs. Dynamic training entails calling the original function for an input instead of performing approximation. The scheme also employs speculative training by calling the original function for selected inputs. Exact outputs and corresponding inputs are stored in the history. This results in the addition of new regions with new polynomials as well as adjustments in existing regions and their polynomials. The scheme can also trigger these changes by updating certain parameters depending on the monitoring info, as will be explained in Section III-D.

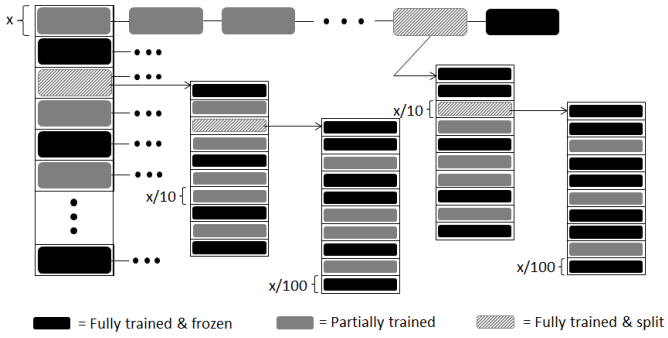

Figure 3. Customized multilevel hash-table. In this example, a bucket is split into a hash-table with 10 elements.

To support run-time improvement, an approximation scheme should use a history data structure that allows dynamic expansion after it has been created. The conventional hash-table used by the basic history-based uniform piecewise scheme does not support formation of new regions or adjustments in region sizes. To support dynamic improvement, PROCsimate introduces a new extensible data structure, described next.

1) Customized Multilevel Hash-table (CMH): PROCsimate introduces modifications to a conventional hash-table to support dynamic expansion. The result is a customized multilevel hash-table (CMH) that allows addition of new regions of smaller sizes by splitting buckets into new hash-tables at deeper levels. Figure 3 outlines the proposed CMH. The first level of CMH is like a regular hash-table. To cover a larger input range, there are lists of history elements that hash to each bucket in the first level of the table. Each history element (interchangeably called a bucket) stores the history for a particular region whose size depends on the level of the table that the element resides in. Each region is further divided into three equal sections (not shown in the figure): top, middle, and bottom. The first input in each section is considered a training input. It is stored at the element with its exact output. After a region has seen three training inputs, it is considered fully trained. Based on the output variation of the training inputs and frequency of use, the scheme decides to freeze or split a bucket. If the output variation is higher and the bucket is frequently used, the scheme decides to split it into a new hash-table with smaller region size. It allows the scheme to have more smaller regions in areas of the function input range where the output variation is high, leading to better approximation. Striped buckets in the CMH of Figure 3, represent the buckets that are split. The region size of a new table is equal to the region size of

the split bucket divided by the size of the new table. In Figure 3, a bucket is split into a new hash-table at a deeper level with 10 buckets. If the output variation for training inputs is less than a threshold, determined by the scheme and tuned over time, PROCsimate decides to freeze the bucket by determining a 0-degree polynomial (constant) as an output for the region. Black buckets in Figure 3 represent frozen buckets. Gray buckets in Figure 3 represent regions that are not fully trained, their output variation is not above the threshold to warrant a split, or they are less frequently used. To approximate such regions, the scheme evaluates the corresponding 2-degree polynomials.

Over the course of the execution, depending on the quality monitoring and frequency of use, PROCsimate may decide to unfreeze previously frozen buckets or freeze or split the gray buckets. If the lists in level-1 of the CMH becomes long or the number of levels becomes large, the scheme can decide to create a bigger CMH, rehash the history from old CMH into the new one, and free the old CMH.

C. Parameter Selection

The history-based uniform piecewise approximation scheme that PROCsimate employs as underlying approximation strategy relies on the following parameters: minimum and maximum input values, size of a region, total number of regions; degree of polynomial, and acceptable output variation. These parameters can affect the quality and performance of the approximation. In the base scheme they are provided by the user who may need to perform multiple trial-and-error attempts and extra profiling runs to find proper parameter setting for an application.

PROCsimate, in its attempt to offer an easy-to-use procedure approximation scheme, does not require the user to provide these parameters. It rather automatically selects parameters based on its observation of the input and output behavior of the function. It tunes them over the course of the application execution based on the changing behavior and approximation quality.

D. Quality Monitoring and Decision Making

PROCsimate monitors the quality of its approximation. It helps the scheme meet guarantees on the accuracy of results that it offers by improving approximations or suspending approximation when ineffective. To monitor quality, PROCsimate uses auxiliary threads that invoke the original function on inputs that the scheme approximated. By comparing the exact and approximated results, PROCsimate guides the performance of the approximation. These decisions, on a high level, include: choosing between approximation versus running the original function on current inputs; doing speculative training or not; policy of determining speculative training inputs; suspending approximation; whether to split, freeze, or unfreeze certain buckets; and whether to rehash. In case of a rehash, the scheme further decides what range of input it should cover and what table size, region size in level-1, and number of buckets in each of the deeper-level tables it should choose.

E. Result Guarantees

PROCsimate offers both hard and soft guarantees about approximation results. It can guarantee that the average percentage error across all invocations of the candidate function will be within the provided error bound. The user provides these parameters (maximum allowed percentage error and whether the user wants hard guarantees or the supplied error bound should be treated as a hint) via command line arguments. To offer guarantees, the scheme checks the approximation results against the exact results using the auxiliary threads that harness available idle cores. In case of hard guarantees, the scheme only approximates the function when it is sure that approximation will not cause the scheme to violate the guarantee. In situations where the scheme cannot be certain, for instance, when the auxiliary threads have pending work and have not been able to find the exact results for earlier approximations, the scheme conservatively executes the original exact function. If the user does not require hard guarantees, the scheme can be more liberal in applying approximation.

The scheme provides guarantees about the result of function approximation. The effect of inaccuracies in function results on the overall execution depends on the application. Some application outputs can only be judged by a human user. For others, a tuning system can be developed, which, for a given error bound on the overall application output, finds the error bound on the function output.

IV. Related Work

Existing function approximation schemes exhibit a number of limitations. Approximate memoization techniques [9], [10], [11] do not employ advanced approximation strategies. For a given input, they return the closest memoized value or at best perform interpolation between two memoized values. The schemes are not dynamic, adaptive, and self-improving. Additionally, the individual approximate memoization schemes lack other features as well. For instance, iACT [11] cannot approximate library functions, does not monitor quality, and requires recompilation of the application to allow different parameter settings. Some other schemes such as [12], [13] can monitor quality and choose from available approximations but require programmers to provide alternate approximate functions, quality metric, and calibration inputs. Requiring programmers' extensive involvement and domain expertise, has its advantages in terms of better approximation in domain-specific settings but limits the adoption and usefulness of the schemes for general applications. Machine learning techniques [14], [15] provide sophisticated approximation strategy, do not rely on programmers' involvement, and are general in their nature,

but require specialized hardware to be of any practical use. History-based piecewise approximation scheme [16] is an efficient, software-only scheme that targets general applications but does not improve approximations dynamically, monitor quality, and offer guarantees. Furthermore, choosing proper parameter settings for an application requires multiple trial-and-error attempts and the uniform scheme depends on the profiling information.

V. Conclusion

Approximating side-effect free procedures can offer significant performance gains to applications amenable to approximation in exchange for some tolerable loss of accuracy. We have introduced PROCsimate, a scheme that approximates procedures. It is a software-only scheme that provides fast and efficient function approximation. The scheme dynamically improves approximation during the application execution, monitors the quality of approximation, and offers result guarantees. PROCsimate automatically selects and tunes its parameters and does not require profiling information or multiple trial-and-error runs by users. It performs dynamic online training and thus benefits applications whose inputs change across runs or even during a single run. Currently, the user has to manually modify an application to insert the code of PROCsimate. In ongoing work, we are automating the code generation. We are also extending the scheme to support polynomials in multiple variables. Lastly, how the function approximation error affects the overall application output is application dependent and not addressed in the current paper. In ongoing work, we are developing a tuning system that finds error bounds for function approximation from application error bounds.

References

[1] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *Computers, IEEE Transactions on*, vol. 54, no. 7, pp. 922–927, 2005.

[2] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 2011, pp. 409–414.

[3] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt, "Exploiting partially-forgetful memories for approximate computing," *Embedded Systems Letters, IEEE*, vol. 7, no. 1, pp. 19–22, 2015.

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

[5] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 25–34.

[6] M. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *ACM SIGPLAN Notices*, vol. 42, no. 10. ACM, 2007, pp. 369–386.

[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 301–312.

[8] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 1–12.

[9] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 102–112.

[10] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 35–50.

[11] A. K. Mishra, R. Barik, and S. Paul, "iact: A software-hardware framework for understanding the scope of approximate computing," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[12] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 198–209.

[13] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice*. ACM, 2009, vol. 44, no. 6.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[15] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "Benchnn: On the broad potential application scope of hardware neural network accelerators," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 36–45.

[16] Aurangzeb and R. Eigenmann, "History-based piecewise approximation scheme for procedures," 2nd Workshop on Approximate Computing (WAPCO), Jan 2016.