A Machine Learning based Load Value Approximator guided by the Tightened Value Locality

Alain Aoun
a_alain@ece.concordia.ca
Concordia University
Montréal, Canada

Mahmoud Masadeh
mahmoud.s@yu.edu.jo
Yarmouk University
Irbid, Jordan

Sofiène Tahar
tahar@ece.concordia.ca
Concordia University
Montréal, Canada

ABSTRACT

This paper addresses two essential memory bottlenecks: 1) memory wall, and 2) bandwidth wall. To accomplish this objective, we propose a machine learning (ML) based model that estimates the values to be loaded from the memory by a wide range of error-resilient applications. The proposed model exploits the feature of *tightened value locality*, which consists of a periodic load of few unique values. The proposed ML-based *load value approximator* (LVA) requires minimal overhead as it relies on a hash that encodes the history of events, e.g., history of accessed addresses, and values that can be extracted from the load instruction to be approximated. The proposed LVA completely eliminates memory accesses, i.e., 100% of accesses, in runtime and thus addresses the issue of memory wall and bandwidth wall. Compared to related work, our LVA delivers a maximum accuracy of 95.16% while offering a higher reduction in memory accesses.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; • **Hardware** → **Memory and dense storage**; **Emerging architectures**.

KEYWORDS

Approximate Computing, Approximate Cache, Approximate Memory, Approximate Load Value, Machine Learning

ACM Reference Format:
Alain Aoun, Mahmoud Masadeh, and Sofiène Tahar. 2023. A Machine Learning based Load Value Approximator guided by the Tightened Value Locality. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3583781.3590207

1 INTRODUCTION

Approximate Computing (AC) has reemerged as an alternative to exact computation in error-resilient applications. AC or *inexact computing* offers an erroneous output for savings in area, power, and delay [8]. AC can be applied to many existing applications such as machine learning, multimedia applications and search engines. These applications tolerate errors due to the lack of a golden answer, noisy and redundant input data, imperfect perception in the human sense of a noisy output, and implementation algorithms with error attenuation patterns. Most of the research in the area of AC has focused on the arithmetic units such as the work in [6]. Approximate arithmetic units are beneficial for computation-intensive applications. Nonetheless, deploying approximate arithmetic units in memory-intensive applications where errors are tolerated, e.g., deep learning, results in minimal performance gain due to the memory wall [12]. Therefore, in this work, we aim to approximate the *memory access* process for load instructions by substituting traditional memory accesses with a machine learning-based predicted value as a way to address this challenge.

Related work targeted *memory wall* with the investigation of process-in-memory, load value speculation and approximate memory, among other techniques. The first approach, i.e., process-in-memory, moves a redundant computational operation, e.g., multiply and accumulate (MAC), from the CPU and place it in the memory or near the memory [7]. On the other hand, load value speculation does not introduce fundamental modifications to the Von Neumann architecture but rather adds a unit in the processor that estimates the value when a load occurs [3, 10, 14]. In case of a wrong speculation, the CPU rolls-back to pre-load, i.e., flushing all instructions executed following a wrong speculation and restoring register values. This idea has been extended to load value approximation where the estimated value is accepted regardless of its correctness, i.e., error distance, such as the work in [15] and [18]. Another approach of introducing approximation to address the memory bottleneck is the work in [11] which introduced an approximate memory. The proposed memory reduces energy of the dynamic random access memory (DRAM) and offers the possibility of loading the most significant bits only. All previous approaches to hide memory latency require access to the memory for quality control.

Machine Learning (ML) has been widely exploited in recent years in various applications. ML can be effective when a given outcome occurs following a series of sequential events [9]. Given the principle of locality in computers, the occurrence of a series of events that potentially result in the same (or a similar) effect has a high probability. The *spacial* and *temporal* localities have been widely exploited in computers. On the other hand, the authors of [10] introduced *value locality* where recently accessed values are correlated. The authors suggest that *value locality* is similar in concept to branch prediction where the outcome of subsequent branches can be correlated to the previous branch outcomes. Subsequently, we investigate the prediction of load values using ML, where we exploit the feature of *value locality*.

In this paper, we propose a ML-based *load value approximator* (LVA) with the aim of addressing the memory wall. The ML model uses the program counter (PC) of the instruction, the effective memory address of a load, and multiple values that encode information about the history of stores and loads as predicates to predict the load value. From our investigations, we found out that Decision Forests (DF) showed superiority to other models such as Neural Networks (NN) and Decision Trees (DT). The tested model achieved a maximum accuracy of 95.25% and a minimum root mean squared error of 19.18.

In the rest of the paper, we will discuss related work in Section 2. In Section 3 we will display observed behaviors when executing load instructions followed by the presentation of the proposed methodology in Section 4. Thereafter, we dedicate Section 5 to analyze the performance of the proposed LVA. We conclude the paper in Section 6.

2 RELATED WORK

Many articles in the open literature have suggested methods to alleviate the memory bottleneck. We will restrict the discussion in this section to those methods that are most relevant to our work. In the sequel, we will present work related to: 1) Load Value Speculation, 2) Load Value Approximation, and 3) Approximate Memory.

2.1 Load Value Speculation

Load value speculation (load value prediction) aims to hide the memory latency by speculating the value to be loaded. One of the earliest works in this area is given in [10] where the authors introduced the idea of *value locality*. The authors suggest that values stored in adjacent memory addresses are comparable in magnitude. For instance, the adjacent pixels of an image stored in memory are alike in value. Subsequently, the authors present a dynamic lookup table that speculates the value of a load with the aim of hiding the memory access latency. In case of wrong speculation, the processor rolls back and flushes the pipeline. Moreover, the lookup table is updated after every memory access. This concept was widely investigated by implementing other speculation techniques such as the work in [3] and [14]. Moreover, all load value speculation techniques access the memory to confirm the correctness of the speculation and roll-back in case of wrong speculation.

2.2 Load Value Approximation

Since roll-backs are expensive in terms of hardware requirement and loss in clock cycles, some researchers have proposed the idea of load value approximation. These techniques speculate a value without a roll-back in case of a wrong prediction which results in approximation. For instance, the work in [15] proposed a load value approximation which relies on a dynamic predictor. The accuracy of the predictor can be improved by providing the value of the recent loads. The proposed model saves in bandwidth and energy for a cost in quality. The work in [18] is another approach to achieve load value approximation where the authors propose a model targeting GPU architecture. Both load value approximation techniques require access to the memory in order to reduce the error.

2.3 Approximate Memory

As an alternative to load value approximation, other researchers have investigated so called, approximate memory, in order to save energy and bandwidth. For example, the work in [11] proposes the implementation of a DRAM where data is stored in a transpose fashion. This would allow each row to have a different refresh rate where rows storing the most significant bits (MSBs) are refreshed more frequently. Moreover, storing the data in a transpose fashion allows reading the MSB of multiple values by a single load. For instance, if a row is 32-bit then executing 22 load instructions would allow the load of the 22 MSBs of 32 values and truncating the other least significant bits. Another example of approximate memory is the work presented in [13] where the authors propose the compression of error-tolerant data in the DRAM. The compression and decompression of different regions of the memory define different accuracy levels. The authors of [13] used a software-hardware integration where the software controls the quality, i.e., compression and decompression of regions. However, both approaches still require access to the memory and thus the memory wall is not fully addressed.

3 PRELIMINARIES

As mentioned earlier, the authors of [10] introduced the concept of *value locality*. They suggested that the values loaded by subsequent load instructions are correlated. In this paper, we examine the behavior of loaded values to identify trends other than the commonly known localities, i.e., *spatial*, *temporal* and *value locality*. To this aim, we propose to instrument the loaded values in a range of applications and kernels from the PARSEC benchmark suite [2]. In the instrumentation, we used *blackscholes*, *bodytrack*, *canneal*, *ferret*, *fluidanimate* and *swaptions* benchmarks with the "simlarge" input [2]. The instrumentation is performed using the GNU Project Debugger (GDB) [5]. Table 1 shows the number of load instructions instrumented per benchmark when executed for the same period. We will base our analysis on the 15,180,241 load instructions gathered from these various benchmarks. In the rest of this section, we limit the investigation to the least significant byte of a loaded value as it is common among various load instructions, i.e., loading 8, 16, 32 or 64 bits. It is noteworthy that the trends presented in the sequel are also observed in the other bytes.

Table 1: Count of Instrumented Loads from Various Benchmarks

Benchmark	# of Instrumented Loads
blackscholes	3,457,447
bodytrack	1,364,347
canneal	4,069,024
ferret	2,896,804
fluidanimate	558,147
swaptions	2,834,472
Total	**15,180,241**

Figure 1 depicts the scatter plot of the loaded values from the memory for the least significant byte (LSB) of the first 200,000 load instructions in the *blackscholes* benchmark. We notice that the value of the least significant byte for the first 12,000 load instructions is incorporating all values in the range, i.e., values from 0 to 255, and a trend cannot be identified. We recognize this portion of the

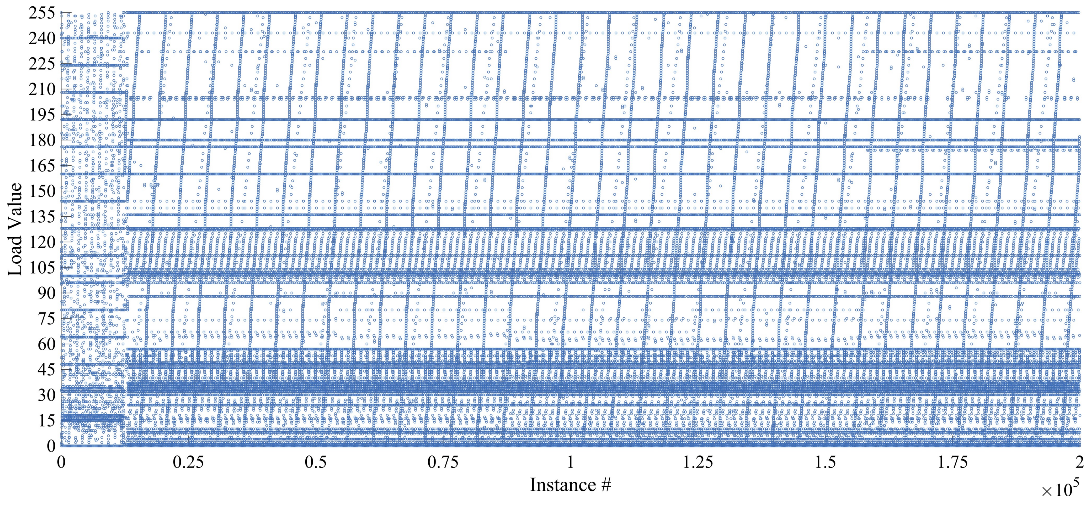

Figure 1: The Value of the Least Significant Byte over Time when Instrumenting *blackscholes*

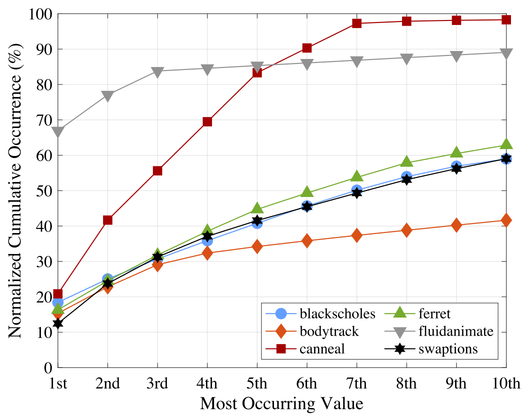

Figure 2: Normalized Cumulative Occurrence for the Most Occurring Values in Various Benchmarks

benchmark as the *cold start*. However, the values of the subsequent load instructions follow a repetitive tendency where we can identify crowded regions, e.g., values between 100 and 125. Moreover, we can notice values that occur with high frequency, i.e., values with scatter plots resembling a straight horizontal line, such as the value of 180. In addition, we can detect a periodic trend in values, e.g., "*almost vertical*" lines appearing periodically. The periodic trend and high frequency of occurring values are observed throughout the instrumented load instructions in *blackscholes*, i.e., beyond the first 200,000 load instructions. Moreover, the aforementioned trends in *blackscholes* are also detected in all other benchmarks covered in this paper, i.e., *bodytrack*, *canneal*, *ferret*, *fluidanimate* and *swaptions*.

In order to comprehend the behavior of values in a program, we calculate the occurrence of a loaded value. Figure 2 shows the normalized cumulative occurrence for the most values loaded from the memory. We can notice that a single value in *fluidanimate* was loaded by 66.93% of the executed load instances, i.e., the first most occurring value. Alternatively, *swaptions* shows the lowest percentage for the first most occurring value with only 12.46%. From Figure 2, we can notice that the minimum normalized cumulative occurrence for the top three values is 29.08%, i.e., almost a third of the load instructions will retrieve one of three unique values. Moreover, the average normalized cumulative occurrence for the top 3 values is 43.71%. On the other hand, the top 10 most occurring values in *canneal* consist of 98.29% of the load instructions. Hence, all the other 246 possible values will be loaded 1.71% of the time. It must be noted that the most occurring values are not unique among all tested benchmarks. For instance, in our instrumentation, the top 10 most occurring load values among the 6 benchmarks consisted of 40 unique values.

Based on the aforementioned observations, we realized that the value of load instructions in a program is periodic and belong to a set of few unique values. We call this concept as *tightened value locality* as it goes beyond the *value locality* proposed in [10] where the authors suggested that a program will load values that are close in magnitude but not unique nor periodic. Subsequently, we suggest building a machine learning-based load value predictor that exploits the *tightened value locality* and hence simplifies its implementation and enhances its quality.

4 PROPOSED METHODOLOGY

Related work has focused on implementing a dynamic predictor where the quality is controlled by increasing the number of effective loads, i.e., actual memory accesses, in order to improve the quality of prediction. On the other hand, given the idea of *tightened value locality*, a conscientiously trained machine learning (ML) model can have the capability of predicting the load values with high accuracy. The ML model can be static and thus eliminates all accesses to the memory in real-time. Since ML models can suffer from overfitting problem [4], i.e., the ML model is trained to include the noise in the training data and thus results in low quality when using a new testing data, a good ML model is expected to deliver high accuracy but with an acceptable level of error. Thus, the proposed ML-based predictor can be implemented in error-tolerant applications (such as image processing or search engines) where an inexact value can be accepted as is and does not require the CPU to

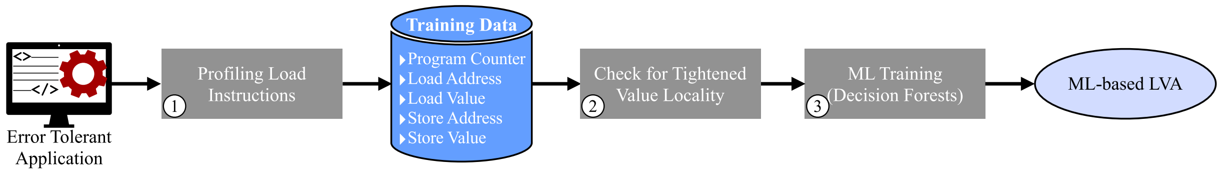
Error Tolerant Application
① Profiling Load Instructions
Training Data
Program Counter
Load Address
Load Value
Store Address
Store Value
② Check for Tightened Value Locality
③ ML Training (Decision Forests)
ML-based LVA

Figure 3: Proposed Methodology to develop the ML-based LVA

roll-back. Subsequently, the proposed ML-based load value predictor is classified as a *Load Value Approximator* (LVA). The proposed LVA can be developed using the three main steps shown in Figure 3, namely, ① application profiling, ② verifying the existence of *tightened value locality*, and ③ building a load value predictor using Decision Forests (DF).

The LVA proposed in this paper requires minimal resource usage to predict a load value, i.e., using either information from the current load instruction or a value encoding history information. The proposed implementation eliminates the use of large tables to store history information. Towards achieving a minimalist predictor, in step ① we profile the targeted application in order to generate training data. For every load instruction, the profiling will record the program counter (PC), the effective memory address, and the loaded value. The PC and effective memory address are needed to differentiate the various load instructions and hence provides the predictor with a local context. The loaded value is needed in order to train the DF model.

During the profiling process, we record the store instructions, where the store address and the stored values are extracted from the instructions and then used to form four hash values. The used hashing is an exclusive OR (XOR) operation. The hash of the store instructions is needed for a static predictor as they provide information about potential changes to the memory and thus allowing a more accurate prediction. The first two hashes are used for local context and computed by XORing the store instances since the last load. For instance, we compute the *local* Hash Store Address (HSA_L) of the i store instructions that occurred since the last load as $HSA_L = Address_1 \oplus Address_2 \oplus \ldots \oplus Address_i$. Similarly, we compute the *local* Hash Store Value (HSV_L), which equals $Value_1 \oplus Value_2 \oplus \ldots \oplus Value_i$. HSA_L and HSV_L are classified as local identifiers as they exclusively encode recent memory modifications that occurred since the last load.

The second set of hashes is *global HSA* and *HSV*, i.e., HSA_G and HSV_G, which are computed by XORing all store instances since the start of the program. HSA_G and HSV_G provide a global context as they encode information related to memory modification since the beginning of the program. On the other hand, providing the LVA with the history of accessed memory addresses, i.e., addresses accessed by previous load instructions, can enhance the decision of the predictor. Thus, we generate a hash by XORing the accessed memory addresses since the beginning of the program. Moreover, we save the address of the last accessed memory. We call the *global* Hash Load Address and the Last Accessed Address HLA_G and LAA, respectively. Subsequently, the generated training data will have one entry for every load instruction where each entry contains six arguments encoding history information, i.e., LAA, HLA_G, HSA_G, HSA_L, HSV_G and HSV_L, along with information about the current load instruction, i.e., PC, the effective load address and the loaded value.

In step ②, the generated training data is analyzed in order to identify if the application incorporates a *tightened value locality*. This feature is detected by looking for a trend in the values loaded from the memory over time, then loading a few unique values with high frequency (such as the example shown in Figure 2). This step is important while implementing the proposed LVA as it enforces the model to remain relatively simple while delivering high accuracy. An application that does not have *tightened value locality* can potentially complicate the ML model. Subsequently, an LVA built based on an application that does not have this property would require high resource usage, e.g., computation power, and thus outlying the benefits of AC.

Finally, in step ③ we train an ML model to construct the LVA. The six arguments encoding history along with the PC and effective memory address are used as predicates in the ML to predict the loaded value. We use ML as it can detect trends in the dataset and generate a model to forecast the outcome when provided a new input data [9]. The the LVA is a static model that eliminates memory access in real-time and can be reused as-is in the repetitive execution of the same application. Neural Networks (NN), Decision Forests (DF) and Decision Trees (DT) techniques are commonly used for ML. Therefore, we limit our discussion to them.

We exhaustively investigated the implementation of these three models and DF was found to be the most suitable for this task. In our analysis, the training of an NN-based model diverged in multiple instances even when using state-of-the-art activation techniques such as the rectified linear unit [1]. Thus the generation of an NN-based predictor may require intensive exploration before finding a suitable model. When deploying the proposed methodology, building an NN-based predictor in real-time might require extensive time and thus outbalance the gains from the implementation of the proposed methodology. On the other hand, since DF is an ensemble of trees and given the natural complexity of the task, DF is expected to outperform DT. This perception was confirmed in our investigation while analyzing the three techniques. The DF-based predictor can be a regression or a classification task when predicting small values, e.g., 8-bit values. For instance, if the prediction is applied to the least significant byte, the classification is achievable since the number of classes is manageable, i.e., 256 classes. However, when building a predictor to predict large values, e.g., predicting the least significant 32-bits, the classification may not be possible and only regression models can be built.

5 EXPERIMENTAL ANALYSIS

We assess the proposed methodology using a range of applications and kernels, namely, *blackscholes*, *bodytrack*, *canneal*, *ferret*, *fluidanimate* and *swaptions* from the PARSEC benchmark suite [2]. The training of decision forests (DF) is performed using TensorFlow

Decision Forests version 0.2.5 [17]. Moreover, the application profiling and quality testing are performed on a machine with Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz running on Ubuntu 20.04. The proposed methodology is applied using various configurations of the DF, e.g., random and gradient DF. The quality of the built LVA is measured in terms of accuracy and the root mean squared error (RMSE). In order to investigate the scalability of the proposed methodology, we generated different LVA models by profiling one application at a time as well as multiple applications. The performed quality analysis is measured for all load instructions while disregarding if the loaded value is part of the control flow of the program. We adopt this approach in order to present the quality of the proposed LVA models when implemented in an environment where the goal is reducing the memory access, e.g., memory bandwidth, to the lowest possible level. In this study, we explored the option of building an LVA with a task of classification and regression. Moreover, in our analysis, we explored the implementation of the LVA as a random and gradient DF. Both DF models resulted in a similar quality. However, we realized that random DF outperforms the gradient DF in terms of training time and size of the model. Thus, we limit the experimental analysis of the proposed LVA to random DF.

We use nine training sets to train 18 DF models to study various quality aspects of the proposed methodology. The first six training sets are generated by profiling each of the six applications separately. Two training sets are generated by profiling two groups of applications, namely, $\textcircled{I}$ *canneal*, *ferret* and *fluidanimate*; and $\textcircled{II}$ *blackscholes*, *bodytrack* and *swaptions*. The two groups are chosen based on the observation in Table 1 where groups $\textcircled{I}$ and $\textcircled{II}$ performed a similar number of memory accesses over the same period of time. Thus, we project the memory accesses to be performed by the two sets of applications to be equal in their execution. Using two groups of applications with a similar number of memory accesses provides a more comprehensive analysis of the proposed LVA.

Finally, we construct one training set by profiling all six applications, i.e., the combination of groups $\textcircled{I}$ and $\textcircled{II}$. Each of the nine training sets is used to train two DF models, namely, Random Forest Classification and Random Forest Regression. The DF models are trained with 300 trees and a maximum depth of 16. Table 2 summarizes the quality achieved for each of the 18 DF models investigated in this paper. From the results shown in Table 2, we can note that when achieving a better accuracy the measured RMSE increased, i.e., lower quality. Similarly, with a smaller value of RMSE the accuracy decreased. Nonetheless, this behavior can be deemed acceptable in approximate computing as the quality is bounded by the rule of fail small or fail rare. Furthermore, in our investigation, we tested advanced training settings in TensorFlow such as using *random seed* and *best first global* growing strategy [16]. However, the advanced training settings offered minimal quality gains and increased the size of the model.

5.1 LVA based on One Application

Using the training sets generated by profiling a single application, we trained various DF models. From Table 2 we can note that the best accuracy was 95.25% and achieved using *ferret*. Similarly, the lowest RMSE was achieved using *ferret* with a value of 19.68. Moreover, we can note that the models based on *ferret*, *fluidanimate* and *swaptions* achieved an acceptable quality with a minimum accuracy of almost 70% when using random forests with a task of classification. On the other hand, the regression-based models achieved a low accuracy with a smaller value of RMSE. Similar to the classification-based models, *ferret*, *fluidanimate* and *swaptions* achieved the best quality with a maximum of 34.93 and an average of 25.04. The models based on *blackscholes*, *bodytrack* and *canneal* failed to achieve an acceptable quality in terms of accuracy and RMSE.

5.2 LVA based on Multiple Applications

Using multiple applications in profiling to build an LVA can be beneficial as the predictor can be used by multiple applications and thus covers a wider spectrum. From Table 2 we can notice that combining multiple applications achieves an acceptable quality. The Random Forest Classification based on group $\textcircled{I}$ achieved an accuracy of 83.41%. Furthermore, the Random Forest Regression predictor for the same group achieves an RMSE of 20.26. The classification and regression achieved an acceptable quality in accuracy and RMSE, respectively. However, the two models resulted in a degraded quality when considering the other metrics. On the other hand, we can note from Table 2 that combining applications randomly may not result in an acceptable quality. For example, the best quality achieved by the models based on group $\textcircled{II}$ is almost half the quality when applications are combined in group $\textcircled{I}$. From Table 2, we notice the quality achieved by the DF model based on group $\textcircled{I}$ is almost the average of the quality achieved by the DF models based on individual applications in the group. However, the quality achieved by group $\textcircled{II}$ is less than the quality of all DF models that

Table 2: Quality of the LVA when using Various Application(s)

Application	Task	Accuracy	RMSE
blackscholes	Classification	61.57%	144.77
	Regression	5.96%	56.56
bodytrack	Classification	53.54%	101.98
	Regression	7.36%	48.23
canneal	Classification	36.57%	135.69
	Regression	7.01%	59.41
ferret	Classification	**95.16%**	169.70
	Regression	0.12%	**19.18**
fluidanimate	Classification	69.14%	110.29
	Regression	7.17%	33.98
swaptions	Classification	81.75%	60.31
	Regression	3.59%	20.48
***Group $\textcircled{I}$**	Classification	83.41%	143.30
	Regression	5.7%	20.26
§Group $\textcircled{II}$	Classification	44.63%	117.95
	Regression	9.58%	42.12
Groups $\textcircled{I}$ & $\textcircled{II}$	Classification	44.65%	117.95
	Regression	7.53%	32.00
Minimum		**0.12%**	**19.18**
Maximum		**95.16%**	**169.70**

**Group $\textcircled{I}$ consists of canneal, ferret and fluidanimate*

§Group $\textcircled{II}$ consists of blackscholes, bodytrack and swaptions

are based on individual applications in this group. Furthermore, based on the Table 1, we notice that *canneal* has the highest number of load instructions when compared to other applications in group $\textcircled{I}$. Thus, we conclude that the quality of the load values in *canneal* were predicted more accurately when combined with other applications. Additionally, combining all applications tested in this paper, i.e., groups $\textcircled{I}$ & $\textcircled{II}$, did not achieve a substantial gain in quality. Based on these experiments, we notice that combining multiple applications has the potential of improving the quality while in others cases it can deteriorate the quality.

5.3 Comparison with Related Work

Previously investigated implementations of load value approximation required constant access to the memory in order to update/train the predictor with the most recent contexts and hence predicting a more accurate value. Subsequently, the accuracy will potentially decrease when reducing the number of memory accesses, i.e., dropping more load instructions. For instance, the authors of [15] experimented with dropping, respectively, 0%, 66.67%, 80.00%, 88.89%, 94.12% of the load instructions that are matching the same entry in the proposed dynamic predictor. Similarly, the work in [18] investigated dropping, respectively, 12.5%, 25%, 50%, 60%, 75%, 80% and 90% of the load instructions. In general, a higher drop rate of load instructions resulted in a lesser quality in the instances presented in [15, 18]. In contrast, our proposed LVA eliminates 100% of memory accesses in runtime, e.g., 15,180,241 bytes were predicted in runtime without accessing the memory while delivering an acceptable quality. For instance, 2,986,804 load instructions are predicted in real-time when executing *ferret* where the average accuracy is 95.16%. Compared to the work in [15], for various drop rates, the error exceeded 20% when the proposed predictor was used to predict the loaded values when executing *ferret*. Furthermore, to showcase the potential of the proposed LVA, the model based on the profiling of the applications in group $\textcircled{I}$ resulted in the elimination of 7,523,975 memory accesses while delivering an average accuracy of 83.41%.

6 CONCLUSION

In this paper, we investigated a machine learning-based *load value approximator* (LVA) with the objective of addressing the memory wall and bandwidth wall bottlenecks. The proposed LVA eliminates 100% of the memory accesses in runtime. The presented methodology profiles a given error-tolerant application(s) to obtain the needed training data. Thereafter, the training data is checked to verify the existence of *tightened value locality*. If the property exists in the training data, then it is used to build the LVA. The proposed LVA demands minimal overhead as the predicates are single values that encode relevant history, i.e., hash value, and information extracted from the current load instruction. The encoding of the history provides the predictor with local and global contexts of the program. The hash values also encode information related to store instructions executed since the last load and program start. The proposed LVA was tested using 8 error-tolerant applications where a maximum accuracy of 95.16% was achieved. Moreover, out LVA can be trained by profiling multiple applications while achieving an acceptable quality. However, not all applications can be grouped as it can result in significant quality degradation.

In future work, we plan to identify the criteria for combining multiple applications to build a LVA with high accuracy. Moreover, we plan to investigate the prediction of load values in fragments to improve quality. For instance, loading 32-bits can be predicted by calling the predictor various times, e.g., prediction in 8/8/16-bit portions. Finally, while the investigation presented in this paper was performed using a x86 processor, i.e., CISC architecture, we plan to extend the analysis to the RISC architecture such as RISC-V or ARM processors.

ACKNOWLEDGMENTS

This research was enabled in part by support provided by Calcul Québec (calculquebec.ca) and the Digital Research Alliance of Canada (alliancecan.ca)

REFERENCES

[1] Abien Fred Agarap. 2018. Deep Learning using Rectified Linear Units (ReLU). *arXiv Computing Research Repository* abs/1803.08375 (February 2018). arXiv:1803.08375

[2] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph. D. Dissertation. Princeton University, USA.

[3] Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. 2006. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Transactions on Architecture and Code Optimization* 3, 2 (June 2006), 182–208.

[4] Tom Dietterich. 1995. Overfitting and undercomputing in machine learning. *ACM computing surveys* 27, 3 (September 1995), 326–327.

[5] GDB developers. 2023. GDB: the GNU project debugger. https://www.sourceware.org/gdb/, Last accessed April 3, 2023.

[6] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. 2012. Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (January 2012), 124–137.

[7] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. 2018. HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory Cube. *IEEE Computer Architecture Letters* 17, 1 (January-June 2018), 5–8.

[8] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. 2020. Approximate arithmetic circuits: A survey, characterization, and recent applications. *Proceedings of the IEEE* 108, 12 (December 2020), 2108–2135.

[9] Michael I Jordan and Tom M Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (July 2015), 255–260.

[10] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 138–147.

[11] Duy Thanh Nguyen, Nguyen Huy Hung, Hyun Kim, and Hyuk-Jae Lee. 2020. An approximate memory architecture for energy saving in deep learning applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 5 (May 2020), 1588–1601.

[12] Geraldo F Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. 2021. DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks. *IEEE Access* 9 (September 2021), 134457–134502.

[13] Ashish Ranjan, Arnab Raha, Vijay Raghunathan, and Anand Raghunathan. 2017. Approximate memory compression for energy-efficiency. In *International Symposium on Low Power Electronics and Design.* IEEE/ACM, 1–6.

[14] Glenn Reinman and Brad Calder. 1998. Predictive techniques for aggressive load speculation. In *International Symposium on Microarchitecture.* IEEE, 127–137.

[15] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *International Symposium on Microarchitecture.* IEEE/ACM, 127–139.

[16] TensorFlow. 2023. API Reference Random Forest Model. https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/RandomForestModel, Last accessed April 3, 2023.

[17] TensorFlow. 2023. Tensorflow Decision Forests. https://www.tensorflow.org/decision_forests, Last accessed April 3, 2023.

[18] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. *ACM Transactions on Architecture and Code Optimization* 12, 4 (January 2016), 1–26.