Low-Cost Prediction-Based Fault Protection Strategy

Sunghyun Park
University of Michigan
Ann Arbor, Michigan, USA
sunggg@umich.edu

Shikai Li*
University of Michigan
Ann Arbor, Michigan, USA
shikaili@umich.edu

Ze Zhang
University of Michigan
Ann Arbor, Michigan, USA
zezhang@umich.edu

Scott Mahlke
University of Michigan
Ann Arbor, Michigan, USA
mahlke@umich.edu

Abstract

Increasing failures from transient faults necessitates the cost-efficient protection mechanism that will be always activated. Thus, we propose a novel prediction-based transient fault protection strategy as a low-cost software-only technique. Instead of re-executing expensive computations for validation, an output prediction is used to cheaply determine an approximate value for a sequence of computation. When actual computation and prediction agree within a predefined acceptable range, the computation is assumed fault-free, and expensive re-computation can be skipped. With our approach, a significant reduction in dynamic instruction counts is possible. Missed faults may occur, but their occurrences can be explicitly kept to a small amount with a proper acceptable range. For evaluation, we build an automatic compilation system, called RSkip, that transforms a program into a resilient executable with the prediction-based protection scheme. Prior instruction replication work shows 2.33× execution time compared to the unreliable execution over nine compute-intensive benchmarks. With a control for the loss in protection rate, RSkip can reduce the protection overhead to 1.27× by skipping redundant computation in our target loops at a rate of 81.10%.

CCS Concepts • **Computer systems organization → Reliability**.

Keywords Reliability, Approximation computing, Redundancy

*This work was done while Shikai Li was a graduate student.

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377920

ACM Reference Format:
Sunghyun Park, Shikai Li, Ze Zhang, and Scott Mahlke. 2020. Low-Cost Prediction-Based Fault Protection Strategy. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368826.3377920

1 Introduction

Over the past decades, researchers have strived to reduce diverse design margins (e.g., noise margin) to maximize performance. However, as tight noise margin is often unable to handle electromagnetic noise properly, computer systems are becoming more susceptible to transient faults which may lead to execution failures [8, 9, 22, 24, 47]. To prevent such failures, diverse techniques introduce redundancy at different levels ranging from hardware to software. Since a transient fault can occur in random location at any given time, the protection strategy must constantly be activated. Also, given that a failure from a transient fault happens rarely at each device perspective, the protection scheme should be fast and cost-efficient to be practical. In general, hardware techniques are powerful but inevitably require design modification and often at high cost. To reduce excessive hardware cost, researchers have proposed software techniques such as Redundant Multithreading (RMT) [14, 23, 29, 39] or instruction duplication under the guidance of the compiler [11, 25, 30, 31]. To provide protection, RMT-based techniques run a redundant thread simultaneously on available built-in resources and validate computations through complex communications between threads. The doubled resource utilization allows for the fast concurrent execution of redundant thread but generally suffer from high energy consumption [19]. Rather than creating a redundant thread, instruction duplication based techniques clone instructions and compare values inside the thread without any additional resources and complicate validation process. Continuous works from Princeton, including the detection strategy called SWIFT [31] and full protection (both detection and recovery) strategy called SWIFT-R [30] [1], propose compiler-directed instruction

[1] SWIFT with TMR-based recovery mechanism.

duplication techniques and provide inspiration to recent explorations [11, 17]. By optimizing instruction duplication, their techniques successfully protect a program with the improved performance overhead. However, slowdown due to increased dynamic instructions remains a concern which needs to be addressed.

This paper proposes a novel prediction-based protection strategy that greatly alleviates run-time overhead of instruction duplication scheme. Without any extra hardware, our strategy focuses on reducing dynamic instructions that must be executed for protection. We postulate that replicated instructions employed by software protection techniques (referred to as *re-computation*) can be bypassed if software approximation techniques can predict fault-free computation result correctly. Otherwise, re-computation must be triggered to check for a possible transient fault. In such cases, mispredictions cause run-time overhead, but not incorrect output like traditional approximate computing techniques [1, 33, 37]. Avoiding any direct impact on the final output of a program, the prediction is used only for validation. This approach may result in missed faults, but we show that their occurrences can be effectively controlled. Sacrificing some protection quality for significant performance improvement is not a new idea [17, 41]. However, our work investigates a different approach in leveraging the trade-off.

We introduce *RSkip*, a prototype of the automatic compilation system that provides the prediction-based fault protection. The system accepts unprotected source code and creates a fast and resilient executable. Since we observed traditional instruction duplication works [11, 30, 31] often suffer at the loop due to its recurring synchronization points, RSkip focuses on improving the protection cost for the loop. As an accurate and lightweight predictor, *dynamic interpolation* is proposed to estimate computation results in the loop by capturing local trends at runtime. A large saving in run-time instruction overhead is possible by replacing the expensive re-computation with a linear equation. As a fallback predictor, *approximate memoization* [33] is also employed.

Throughout the paper, a *Single Event Upset* (SEU) fault model is adopted and the memory system is assumed to be protected given that the commercial DRAMs and caches equip Error Correction Codes (ECC) [31]. Our major contributions are as follows:

- We propose a novel prediction-based protection strategy and demonstrate that software approximate computing techniques are not limited to performance optimization in applications that can tolerate output error. Rather, they can be effectively adapted to detect transient faults in a cost effective and accurate manner. Although this work demonstrates the effectiveness of our new protection strategy by using two loop output estimation techniques targeting several popular computation patterns, its applicability can be broaden with new approximation technique that has a wider target.
- By leveraging the acceptable loss in protection rate [2], our work reduces the number of dynamic instruction greatly to 42.82% while alleviating overhead 1.8× compared to the conventional approach.
- Two approximation techniques are adopted as the prediction model: *Dynamic Interpolation* and *Approximate Memoization*. We newly propose dynamic interpolation that utilizes redundancy for accurate low-cost value prediction. Also, performance and efficiency of approximate memoization are improved from the previous work [33]. With both models, RSkip can bypass 81.10% of re-computation in our target loops.
- Run-time management system is inserted to handle run-time dynamics (e.g., diverse inputs). *Context signature* is suggested to diagnose the current situation and give an indication to tune approximation aggressiveness.

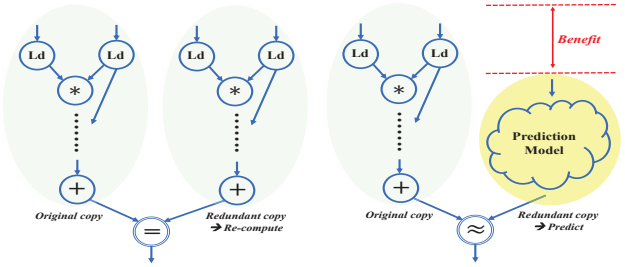
2 Motivation and Idea

Conventional instruction duplication techniques [11, 30, 31] replicate the exact same sequence of instructions for fault detection [3] and compare computed values at synchronization points to identify a mismatch which can be a transient fault. Synchronization points include *store*, *branch* and possibly *function calls* depending on the implementation. If a fault is detected, a recovery mechanism will be triggered. Because of the extra instructions for re-computation and validation, previous approaches often suffer from high run-time overhead that may not be feasible in many situations (e.g., mobile device). Thus, we consider alternatives to reduce overhead.

Figure 1 illustrates the overarching idea of our prediction-based approach in comparison to conventional approaches. Two strategies adopt distinct approaches toward managing redundant copies of instructions for verification. In the figure, recovery routine is omitted for simplicity.

Figure 1a highlights the concept of the conventional instruction duplication techniques. Even without considering extra overhead for recovery, the previous detection techniques perform more than 2× dynamic instructions as the unprotected program due to extra instructions for duplication and validation [4]. As a result of parallelism inside modern processors, slowdown of conventional detection techniques is reported less than 2× [31]. Yet implications suggest that previous approaches would suffer from a large overhead when it is unable to fully utilize parallelism to cover increased dynamic instructions. For instance, periodic reaching of synchronization points adds dynamic instructions with dependencies for validations, often causing previous approaches

[2] 5 percentage point loss is considered acceptable. See Section 7.3 for details.
[3] Recovery requires additional instructions.
[4] With recovery mechanism, the full protection scheme may need to execute more than 3× [30].



(a) Conventional **(b)** Prediction-based

Figure 1. Idea of each protection strategy. **(a)** Re-compute and validate. **(b)** Estimate and fuzzy-validate.

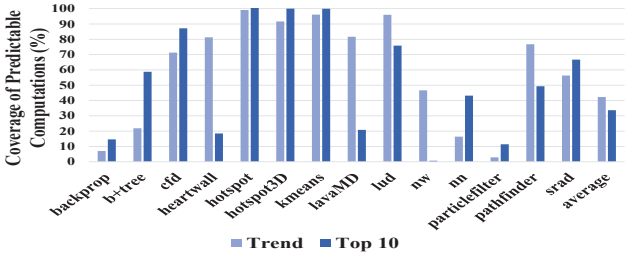


Figure 2. Proportion of dynamic instructions whose computation outputs can be estimated. Measured in Rodinia [10] benchmark suite.

to struggle at the loop. Given that compute-intensive programs consume significant amount of time on the loop, it can be a serious drawback. This phenomenon will be discussed further with the experimental data in Section 7.

Rather than replicating identical chains of instructions, prediction-based approach estimates computation results by using software approximation techniques as shown in Figure 1b. If prediction and original computation agree within the predefined acceptable range, the original computation is assumed correct and used to proceed the rest of program execution without re-computation. In this work, *relative difference* is used to define acceptable range. Naturally, benefit will be the cost gap between re-computation and prediction model. Otherwise, a value is considered as a perturbation which may be a possible fault as in perturbation screening [26, 32]. In such a case, re-computation will be triggered for the exact validation. **Thus, mispredictions (false positives) result in run-time overhead without having a direct impact on the program output since estimations are only used for validation.** Traditional approximation technique requires high prediction accuracy as it replaces the original computation and thus directly influences program output quality. However, in our approach, prediction accuracy requirement can be relaxed: we only need enough accuracy to recognize perturbation at runtime.

Due to the existence of fuzzy validation, a small error may avoid fault detection. A fault must satisfy two conditions to avoid prediction-based fault detection and cause a failure. First, a transient fault should modify the value in the original copy within the acceptable range. Otherwise, correction will be invoked. Note, a fault occurred in the redundant copy does not influence the final output of the program since the copy is only used for validation. **Therefore, missed faults (false negatives) can occur, but their occurrences can be explicitly kept to a small amount with proper acceptable range. With reasonable acceptable range, we can effectively control the occurrence of false negatives with consideration for the trade-off for the significant performance improvement.** To set a reasonable acceptable range, the occurrence of false negatives is investigated in Section 7.2. Secondly, the corrupted value must be fatal to program fidelity. To avoid this situation, we do not apply approximation on particular types of values, such as pointers or loop induction variables, that can significantly impact program correctness. Thus, they are protected with traditional instruction duplication. Usually, their impact on performance is marginal due to their low computational overhead.

To maximize benefit from prediction-based protection, a approximation model should naturally be lightweight and accurate. Since prior works struggle from high protection overhead over the loop on which a compute-intensive program usually spends significant time, we choose the loop as the main optimization target and design specialized prediction models for the loop. To find an opportunity, we investigate outputs of major loops in Rodinia benchmark suite [10] and observe the following phenomena: (1) outputs produced in consecutive iterations tend to share a certain trend. (2) since the same computation is conducted repeatedly in a loop, there may exist many repeating outputs. A trend in loop outputs occurs along with *spatio-value similarity* [35] that arises when data elements with spatial locality tend to be inherently consistent. If a trend can be captured accurately, it would be possible to efficiently estimate outputs that require a significant amount of computations. Compared to previous work [35] that simply groups nearby similar values, trend-based prediction can be more effective way of utilizing *spatio-value similarity* by having a wider coverage (values on the same trend might not necessarily have close values.). Likewise, frequent output values can be used for prediction. Figure 2 suggests the potentiality of such approaches. We measure prediction accuracy of each method and reflect impact of predictable computations. For the trend-based prediction, data elements showing less than a certain amount of changes in consecutive iterations are considered residing in the same trend. However, sometimes, a few outliers irritate the trend-based prediction. In this motivational experiment, we manually handle those corner cases. Also, we examined the effectiveness of a prediction method that only uses the top 10 most frequent values. Our motivational experiment shows that both prediction models present promising result, implying the chance of bypassing more than 33% of dynamic instructions of the entire program.

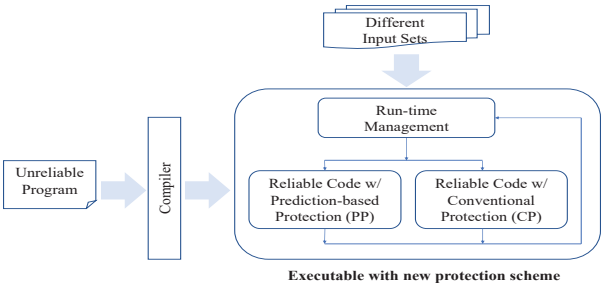


Figure 3. System Overview

```
1  for(i=start;i<end;i++){
2      price = BlkSchlsEqEuroNoDiv(sptprice[i], ...);
3      // Re-computations for 'price' will be
           substituted.
4      prices[i] = price;
5  }
```

(a) Detected pattern in *blackscholes*: function call

```
1  for(j=i+1;j<size;j++){
2      sum = a[j*size+i];
3      for(k=0;k<i;k++)
4          sum -= a[j*size+k]*a[k*size+i];
5      sum = sum/a[i*size+i];
6      // Re-computations for 'sum' will be substituted.
7      a[j*size+i] = sum;
8  }
```

(b) Detected pattern in *lud*: loop

Figure 4. Detected patterns in real benchmarks

However, existing techniques are discovered to be inaccurate or relatively expensive in capturing varying trends. A regression analysis, for instance, requires a process of splitting data elements to determine the coverage of a single equation. In general, however, such a split process is out-of-scope for a regression analysis [21] and naive split policy would result in incorrect estimations due to the tendency of total dependence on the input. Also, a regression analysis may require high training overhead since training should be done for each equation separately. To overcome these challenges, *dynamic interpolation* is proposed. Dynamic interpolation learns how to split data elements to capture varying trends. Rather than learning each equation during offline training, dynamic interpolation computes a linear equation at runtime by using two endpoints in each split and use the equation to estimate values between those endpoints.

Although dynamic interpolation can capture rapidly changing short trends (e.g., data with low spatio-value similarity), there is an upper bound for skipping re-computation as interpolation cannot estimate values for endpoints. To skip re-computations further, we introduce *approximate memoization* [2, 3, 33] as a second-level predictor. Without dependence on trends, the method substitutes expensive computation, such as the function call, with a single access to a lookup table that stores popular repeating values.

Since approximate memoization is generally more expensive than dynamic interpolation, the first prediction will be made by dynamic interpolation. When the interpolation turns out to be wrong, approximate memoization creates a second prediction. Yet, this is profitable as the overhead for two consecutive predictions can still be lower than the overhead of the expensive re-computation. In *blackscholes*, the relative cost between dynamic interpolation, approximate memoization, and re-computation measures 1:1.84:4.18 justifying our approach.

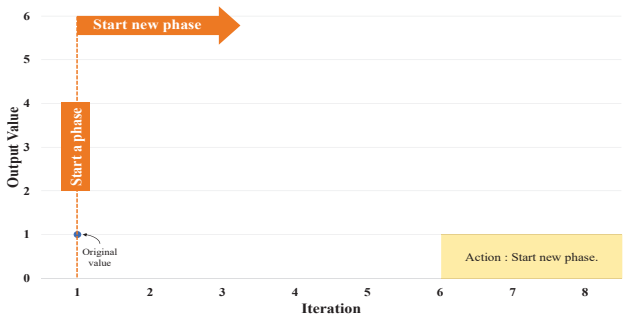
3 System Overview

RSkip is a prototype of fully automatic compilation system that provides prediction-based protection. The system takes unreliable source code as an input and generates a lightweight resilient executable. It requires no hardware modification and no preprocessing on source codes or input data sets. [5]

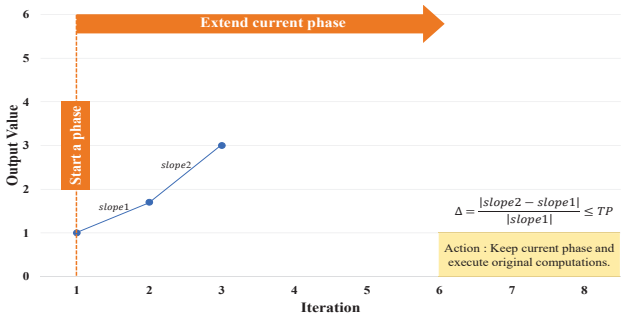
Figure 3 outlines the concept of RSkip. By exploring source codes, the compiler conducts a thorough static analysis (e.g., def-use chain) and detects optimization candidates. Since the current prototype focuses on the loop, the compiler identifies the loop that available approximation techniques target. Once a target loop is isolated, two different versions of the loop will be created: a *reliable version with prediction-based protection (PP)* and a *reliable version with conventional protection (CP)*. Later, at execution time, one of two versions will be chosen by the run-time management which is designed to handle the run-time dynamics. CP will be chosen in cases where PP is expected to have no benefit. Code regions which are not selected as the optimization target will be transformed into the CP without PP and interaction with run-time management.

After one-time compilation, the executable starts the offline training process. It observes the run-time context (e.g., trend pattern for dynamic interpolation) to recognize the current situation and learns how to adjust the approximation aggressiveness. Run-time context can be altered when the program runs with different input sets or the same code is executed with different live-in values within a single run. To react such a run-time dynamics, run-time management uses the run-time data to create a *context signature* for each of transformed code regions with PP. A signature depicts the run-time context and is used to adapt approximation methods. Section 5 describes further about context signature and run-time management.

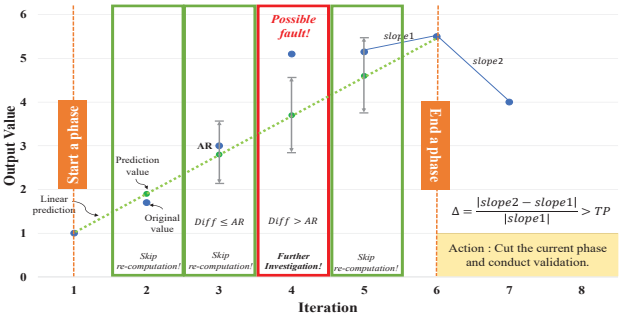
[5] In instances where an user desires the highest protection rate in a specific code region, the acceptable range can be specified as zero by using pragma. Otherwise, a default acceptable range will be applied.



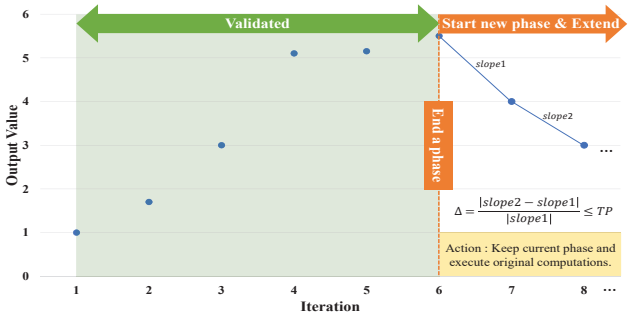
(a) 1st iteration: Start a new phase.



(b) 3rd iteration: Extend the current phase.



(c) 7th iteration: Cut the current phase and conduct validation.



(d) After 7th iteration : Repeat (b) - (c)

Figure 5. Sketch of dynamic interpolation. Whenever slope change is above tuning parameter (TP), a phase is defined. During the validation process at (c), a data element is considered as a possible fault if the difference between original value and prediction value is greater than acceptable range (AR).

4 Prediction Techniques

As previously discussed, we chose approximation techniques specialized to the loop. Since *store* is a synchronization point which often requires heavy computation for its value operand, our approximation techniques estimate values that will be saved in the memory within the loop. However, loops with certain types of value computation (e.g., pointer) or low computation overhead (e.g., initialization) are not considered as the approximation target loop. These will be filtered out by the static analysis with the cost estimation. For simplicity, we target the legitimate types of value computation containing the loop or the user function call that has the number of instructions above threshold as in Figure 4. The current approximation target may include data that is not generally considered as approximable. However, since the approximation would be used only for validation, it would not harm the protection quality until a missed fault occurs on the corresponding code segment in the original computation. The missed fault would impact the protection quality according to its influence on the program fidelity, thus our experimental result reflects this factor.

4.1 Dynamic Interpolation

4.1.1 Target Computation Pattern

Dynamic interpolation targets computation that updates values in the memory at every iteration. Either the expensive function call (Figure 4a) or computation with the loop (Figure 4b) can be replaced with a single linear equation. Dynamic interpolation also works on multi-dimensional array. For example, although Figure 4b had written with multi-dimensional array, dynamic interpolation would still be applied.

4.1.2 Implementation

Skip rate (i.e., the ratio of iterations skipping re-computation in the loop) of dynamic interpolation largely depends on how a phase (i.e., consecutive data elements that a single equation covers) is sliced onto data elements to capture local trends. Particularly, dynamic interpolation should be intelligent enough to maximize stride (i.e., phase length) on a long trend. When values fluctuate widely, it should lower stride. In general, due to the tendency of total dependence on the input, phase cannot be determined statically. To overcome the challenge, we notice the opportunity to utilize the redundancy inserted for protection mechanism: a copy of the computation can be used as run-time guidance to cut the phase of a redundant computation dynamically.

Figure 5 demonstrates the dynamic interpolation algorithm step by step. To decide approximation aggressiveness, we introduce a tuning parameter (TP), which represents optimistic expectation towards outliers on phase slicing. With a higher TP, the algorithm tries to extend a stride more aggressively by ignoring outliers. Although this example assumes a TP is already known, in reality, it will be properly adjusted by the run-time management. More details are explained in Section 5. Figure 5a shows the setup stage for a new phase. Once the first point is generated, the algorithm proceeds to next iterations to see if an extension is possible. Figure 5b presents the extension stage. If a change in the latest two slopes is less than TP, a point is considered to reside in a current trend. While the condition satisfies, the algorithm keeps the current phase and proceeds to the next iteration. Up to this point, only original computations are performed and their results are saved in their original memory space without allocating any extra space. In case that the loop reads and updates same memory locations (e.g., A[i]+=1), we allocate temporary space to keep the original value and use it for re-computation. Although managing temporary space occurs some overhead, our evaluation shows significant improvement is still possible (See benchmark *lud* in Section 7.1.). If a slope change is above TP, the phase is cut at the previous iteration. The cut stage is described in Figure 5c. When a phase is defined, data elements in the phase are validated with a linear prediction computed by two endpoints. If the original computation and the prediction agree within the acceptable range, the algorithm assumes fault-free and bypasses the re-computation. Otherwise, it suspects a possible fault and triggers further investigations. Once the validation for the current phase is done, the next phase starts. In this case, the setup stage is no longer necessary. Therefore, the extension stage and the cut stage are repeated after iteration 7 as shown in Figure 5d. Note, our work did not set any upper bound for a stride.

4.2 Approximate Memoization

4.2.1 Target Computation Pattern

To apply this method, the computation should generate the identical output on the same input set without any side effect (i.e., I/O execution). Also, the number of inputs should be reasonably limited as lookup table size is expected to grow exponentially. These strict requirements force the technique to have narrower applicability than dynamic interpolation.

4.2.2 Implementation

Due to the limited memory space, a lookup table cannot contain all possible cases. Therefore, approximate memoization will group the nearest inputs through the quantization process and make them access the same entry in the lookup table. Naturally, its accuracy and performance entirely depend on the table size and the quantization-based address calculation. In general, a larger table shows higher accuracy, but its size should be determined with consideration for both memory size and performance benefit. An overly large table may not fit in the cache and induce complexity to the quantization process along with the address calculation. Given that the technique should perform the quantization-based address calculation for every inference, its complexity will affect access time towards the memorized result. If the prediction target is computed by many inputs with a diverse spectrum of values, the lookup table size needs to grow large enough to cover a variety of input combinations while being able to return saved results faster than original computation time. Therefore, it is important to find an intelligent lookup table management strategy and the quantization becomes a key process in lookup table construction. In general, the construction algorithm determines the number of quantization levels for each input by assigning a certain number of bits in the address bits. Since the width of the address bits is decided by the lookup table size, the smart management strategy should be able to distribute the limited number of bits to each input while maximizing prediction accuracy. To overcome this challenge, Samadi et al.[33] propose the systematic quantization method with the bit tuning process. By allowing more bits, the technique enables inputs with higher impact on the final output to better differentiate their input sets. After bit tuning, both minimum and maximum values for each input are used to determine the coverage of each quantization level. With the assumption that inputs are uniformly distributed, they equally divide the region between the minimum and maximum values into the assigned number of quantization levels. But, when inputs do not follow a uniform distribution, significant inefficiency may arise with this approach. Therefore, in this paper, the coverage of each quantization level is dynamically determined based on the profiling analysis. For dynamic determination, we first build a histogram with narrow-ranged uniform-length bins and gradually combine nearby less-crowded bins. It becomes apparent the new approach is able to build a more efficient lookup table than the previous work. At *blackscholes*, the previous approach encodes three inputs out of six with 15bit-wide address. Upon the same function, our approach encodes six inputs with an equal width of the address. Naturally, accuracy is improved from 96.5% to above 99% when testing with given representative inputs in the benchmark suite [6]. Once compiler statically identifies candidate loops, lookup tables will be constructed and examined during training phase. If the lookup table shows good prediction accuracy with training data, it will be deployed at runtime. However, when run-time performance is not as good as expected, run-time management may disable the deployment.

5 Run-time Management

Since traditional approximation techniques replace the original computation, Quality-of-Service (QoS) management is a key process to guarantee a certain level of output quality

Table 1. Selected benchmarks. The impact of skipping re-computation can be imagined by provided computation type and location. Both training input and test input are randomly generated or selected without any intersection.

Benchmark	Application domain	Description	Computation type of prediction target	Location of detected loops	Input
conv1d	Signal processing, Machine learning	1D convolution	A reduction loop	Inside a outer loop	4048 integers
conv2d	Signal processing, Machine learning	2D convolution	Nested reduction loops with conditional statement	Inside a outer loop	200*200 integers for input vector, 15*15 integers for kernel
sgemm [40]	Linear algebra	General matrix multiplication	Nested reduction loops	Inside a outer loop	1024*1024 integer matrices
kde [20]	Machine learning	Kernel Density Estimation	Nested reduction loops	Inside a outer loop	1500 float vector
forwardprop [10]	Machine learning	Forward propagation for the fully connected neural network	A reduction loop	·	1024*1024*1024 network
backprop [10]	Machine learning	Backward propagation for the fully connected neural network	A reduction loop	·	1024*1024*1024 network
blackscholes [6]	Finance	Stock price prediction model	A function call	Inside a outer loop	65536 cases
lud [10]	Linear algebra	LU decomposition	A reduction loop with a varying trip count	Inside a outer loop	1024*1024 float matrices
YOLOv2 [27, 28]	Machine learning, Computer vision	Real time object detection	A reduction loop	Inside a outer loop	0.1 ~10MB images

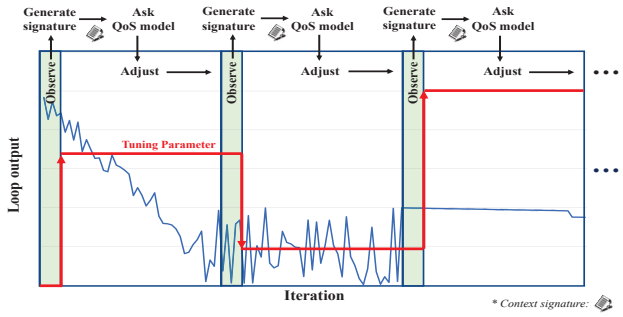


Figure 6. An example of run-time management for dynamic interpolation. Run-time management periodically generates context signatures by summarizing the current run-time context to adjust TP based on the QoS model.

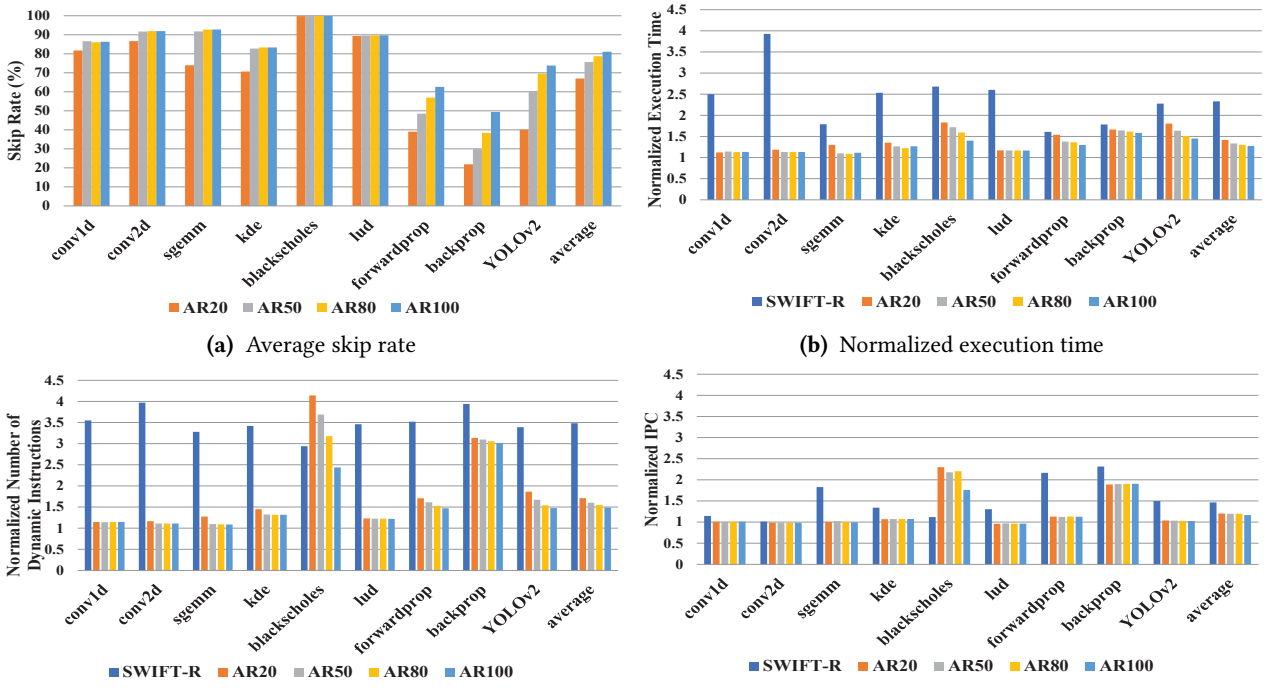
while providing significant performance improvement [5, 18, 34]. Green [5] is the representative framework that guarantees QoS of approximation techniques at runtime. Before its deployment, Green constructs a QoS model at offline by using user-provided inputs. Later at deployment, the framework observes the run-time context and adjust approximation decisions accordingly with its QoS model. To handle input diversity, we follow a similar approach. In our work, the prediction accuracy of each approximation method is adopted as QoS metric. To provide high QoS, a *context signature* is defined to provide each approximation technique with a meaningful summary of the run-time context. For dynamic interpolation, the statistics of local trends can be a useful indicator reflecting the run-time context. In our experiment, we generate a signature by using histogram of slope changes which implies the impact of TP. For example, signature "312" means 3rd bin has the largest count followed by 1st bin and 2nd bin. Since the run-time context may change during an execution, the management system periodically triggers observation and adjustment processes. Thus, an executable may generate more than one signatures during its execution. Figure 6 illustrates how the run-time management adjusts a parameter for dynamic interpolation by using the context signature. Blue line indicates output values across iterations. Note, TP needs to be adjusted based on the slope changes rather than the number of outliers. Since our dynamic interpolation algorithm will split phases into two at the outlier when its divergence is greater than TP, a large TP is preferred on a big trend to ignore small outliers. Therefore, our QoS model will escalate TP in a long trend to extend the phase aggressively. In contrast, the parameter should be decreased in widely-fluctuating short trends to avoid wrong predictions. The QoS model will monitor the prediction accuracy at run-time and may disable the dynamic interpolation at low accuracy. However, we could not observe this case in our experiment. Since approximate memoization does not depend on the parameter, its QoS model simply monitors the occurrence of misprediction and disables its usage at poor run-time accuracy.

6 Training

During the offline training phase, RSkip will build prediction models and construct their QoS models. By utilizing the training set provided by users, RSkip samples outputs from detected loops. For dynamic interpolation, RSkip simulates its algorithm on samples by sweeping various parameters and monitors performance (e.g., skip rate) to identify the best parameter for each signature. Note, we "simulate" algorithm (i.e., phase-splitting and prediction) without repeatedly running a real program to minimize training time. Once the best parameter is identified, RSkip builds a QoS model which includes a table containing (signature, best parameter) pairs. Later at runtime, RSkip simply reference this table and load the learned parameter when a signature is found. Otherwise, we kept the previous tuning parameter although different policies can be chosen. In addition, the lookup table will be built by following our construction algorithm discussed in Section 4.2. Then, during the inference phase, approximate memoization makes predictions by simply reading memorized results in the lookup table.

7 Evaluation

As described in Section 4, our prototype implementation of RSkip targets a computation containing the loop or the user function call as a candidate for prediction-based protection scheme to evaluate its potential. When a program does not



(a) Average skip rate

(b) Normalized execution time

(c) Normalized number of dynamic instructions

(d) Normalized IPC

Figure 7. Test result with test inputs of each benchmarks (AR : Acceptable Range)

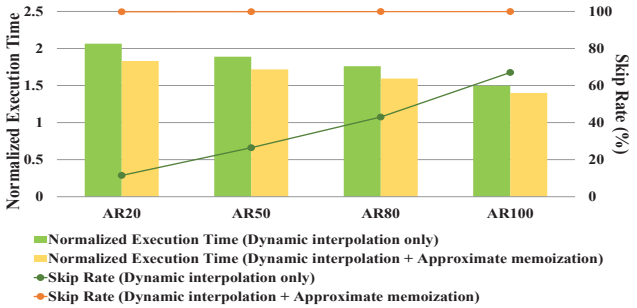
contain such computations, they are protected solely by the conventional protection scheme. Therefore, nine applications with target computation types are chosen as benchmarks from various domains. For a case study, diversity of detected patterns is also considered during benchmark selection. The selected benchmarks represent compute-intensive applications containing dominant loops. Especially, *YOLOv2* is a successful real-life object detection application. Table 1 explains characteristics of each application. To ease preparation effort, both training inputs and test inputs are randomly generated by using the input-generator or selected from well-known image archives without any intersection between them. If inputs can be selected manually, better training quality would be expected. Throughout the experiment, 20%, 50%, 80%, and 100% acceptable ranges [6] are assumed for RSkip (labeled as AR20, AR50, AR80, and AR100 respectively). The rationality of the acceptable range will be discussed in Section 7.3. Note, approximate memoization is only applied to *blackscholes* due to its strict requirement as previously discussed in Section 4.2. Without focusing on the improvement of the recovery mechanism, the re-computation based recovery technique is chosen in this work and its impact is reflected in our analysis. Since fault detection and fault recovery mechanism can be investigated independently [17, 31], a better recovery mechanism can be incorporated if applicable. To handle interaction with the unmodified library function, both schemes set the function call as the synchronization point. As baseline, SWIFT-R [30] is chosen since it is one of the most well-known instruction duplication techniques that provide full protection.The execution is forced to use a *single thread* and the experiments are conducted after an automated training session with training inputs. The extension for multi-threaded execution remains as the future work.

[6]Relative difference is used in this work.

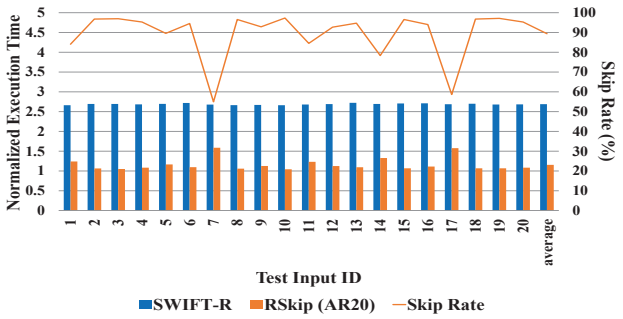
7.1 Performance Overhead

The performance of each protection scheme is measured by running benchmarks on an Intel Xeon CPU E31230 with 3.20GHz. Also, *papi* library [44] is used to measure the number of dynamic instructions and Instructions Per Cycle (IPC).

Figure 7 shows the performance with test inputs for all benchmarks. Figure 7a presents the ratio of iterations skipping re-computation in the detected loop. Overall, 67.03% of re-computation can be bypassed with the value prediction at AR20. With a wider acceptable range at fuzzy validations, the skip rate increases steadily with AR50 (75.67%), AR80 (78.73%), and AR100 (81.10%). The performance overhead is also presented in Figure 7b. **The entire program execution time of each protection scheme is normalized by the execution time of the unprotected program.** On average, SWIFT-R suffers from 2.33× slowdown due to repeating synchronization points. A 20% acceptable range shows a 1.42× slowdown with 67.03% skip rate. The performance also improves due to the increasing skip rate as the acceptable range grows. When the acceptable range is set 100%, the slowdown is measured only 1.27× with 81.10% skip rate. By using



(a) Test result in *blackscholes*.



(b) Test result of AR20 in *lud*.

Figure 8. The detailed analysis for two selected benchmarks

skip rate, frequency of recovery mechanism can be rougly estimated. To further understand the experimental results, the normalized number of dynamic instruction and IPC for the entire program execution are presented in Figure 7c and Figure 7d respectively. SWIFT-R executes 3.48× dynamic instructions while having only 1.47× IPC improvement. As the increase of IPC is not enough to hide the additional instructions for protection, the technique suffers from slowdown. On the other hand, RSkip achieves performance improvements by inserting significantly fewer instructions while having a similar level of IPC with an unprotected program. AR20 inserts only 1.71× additional instructions which can be decreased further by 1.49× at AR100.

The performance of SWIFT-R is degraded by a large amount in *conv2d* as the value is calculated in nested loops with conditional statements inside them. Because of recurring synchronization points in the code with a complicated control flow, SWIFT-R cannot exploit the hardware parallelism well enough so that its IPC is almost the same with the unprotected program. At the same time, it executes 3.97× extra instructions for repeating validations. In this case, the benefit of skipping re-computation is significant. By skipping 86.67% of re-computation, AR20 shows the best-averaged performance improvement in *conv2d* among selected benchmarks. And the performance can be improved further with broader acceptable ranges.

blackscholes shows the highest skip rate, which is above 99%, with every acceptance rate due to the existence of approximate memoization as a second-level predictor. Interestingly, even without the significant change in skip rate, the performance improvement is still observed when the acceptable range grows. To understand this phenomenon, the detailed analysis is conducted for this application as shown in Figure 8a. The figure evaluates the presence of the fallback predictor by comparing both normalized execution time and skip rate. Without approximate memoization, AR20 shows 2.07× runtime overhead with 11.47% skip rate and the gradual improvement is observed as the acceptable range broadens. With 100% of acceptable range, the skip rate of dynamic interpolation increases up to 67.03% resulting in 1.50× slowdown. This can explain the gradual performance improvement of the RSkip with the presence of secondary predictor. When the re-computation is skipped at the secondary predictor, the estimation cost includes the overhead of the first prediction as well as the second predictor. Although, the cost is cheaper than re-computation, it is quite expensive given the relatively high cost of approximate memoization. Thus, the contribution of the first-level predictor in the skip rate has a noteworthy impact. Consequently, as shown in Figure 8a, the performance of RSkip [7] can still improve even among the similar skip rates due to the increase of the contribution from the first predictor in the skip rate with broadened acceptance rate.

To assess the impact of input diversity, the variance in performance and skip rate towards different test inputs are observed with AR20. Figure 8b presents a representative result with *lud*. Mostly, performance and skip rate are measured close to 1.15× with 90.00% respectively. The worst case is measured as 1.59× slowdown with 55.00% skip rate. Yet, significant enhancement from SWIFT-R is observed. The best case is measured as 1.07× slowdown with 97.15% skip rate. In this case, the overall run-time overhead for full protection scheme (detection + recovery) is only 7% of unprotected program execution.

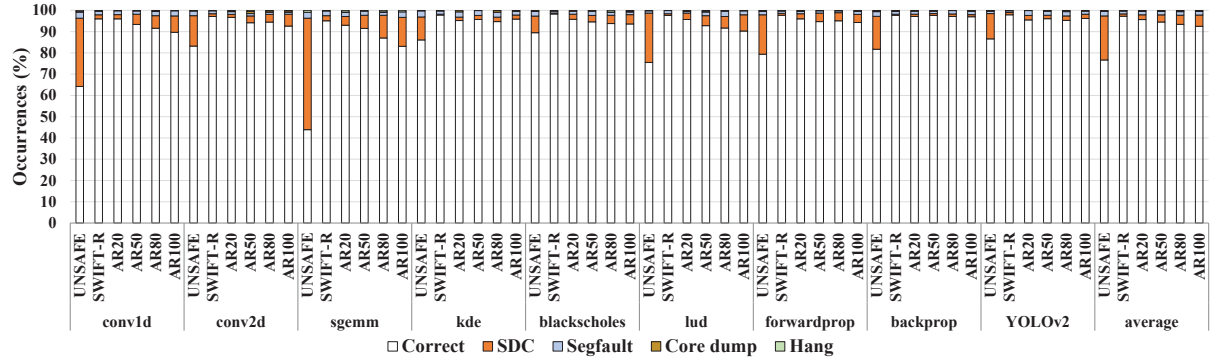
7.2 Reliability

The reliability aspect is evaluated using *Statistical Fault Injection* (SFI) [12, 17] in Gem5 simulator [7]. Each of nine benchmarks is executed 1,000 times on the out-of-order cores with a configuration of ARMv7-A. By following the previous work [17], a single bit flip is injected randomly into the simulated processor during each run. **In our reliability evaluation, faults are only injected into the detected loops to strictly evaluate the reliability of prediction-based protection approach. Note that, in real-life execution, a fault can occur at regions outside of detected loops that are protected by the traditional approach.**

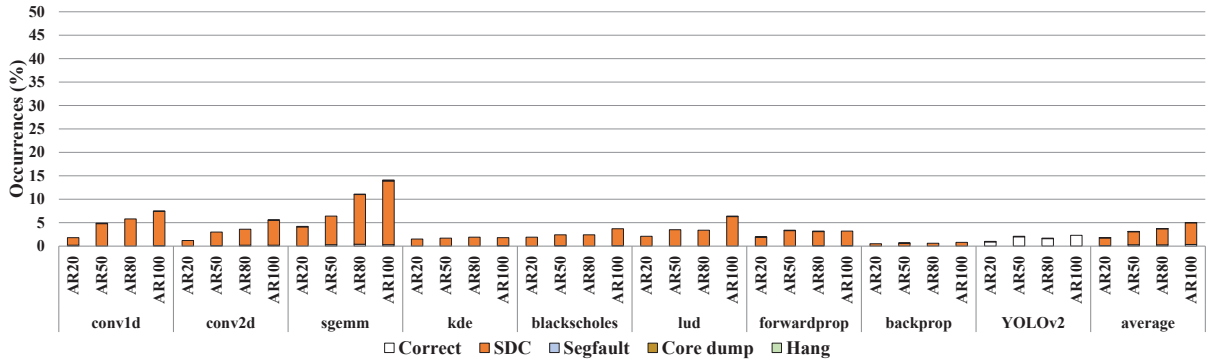
The simulation result is categorized into five classes:

- Correct: The execution generates correct output without any data corruption. Conventional approximation

[7] RSkip employs both dynamic interpolation and approximate memoization



(a) Fault injection experiment : 1,000 faults are injected for each benchmark.



(b) Measurement of false negatives.

Figure 9. The result of the fault injection experiment. Each protection scheme is tagged under every application.

methods often ignore certain amount of error [1, 33, 37] in the output quality. **However, our evaluation considers even small output errors as bad quality and only 100% of output quality as "Correct".**

- Silent Data Corruption (SDC): An influence of single bit flip silently remains until the program termination and creates corrupted output.
- Segfault: Failure due to illegal memory access.
- Core dump: The injected fault results in a system crash or an abnormal termination of the program.
- Hang: The program cannot terminate its execution.

Figure 9 shows the result of fault injection experiment. Fault protection rate of each protection scheme is provided in Figure 9a. For comparison, the unprotected program is included and labeled as UNSAFE. On average, 76.68% of injected faults are masked in UNSAFE. Additionally, they suffer from 20.72% SDCs and 2.13% Segfaults. SWIFT-R shows a 97.24% protection rate, with the decreased occurrences of SDCs and Segfaults to 1.08% and 1.40%, respectively. Such failures are caused by limitations of the software-only protection scheme [31]. Since there is no dedicated mechanism to protect special registers, detection is not possible when a transient fault strikes a single bit in the opcode field and changes the instruction into operations like *branch* or *store*. Also, a transient fault may occur at the examined register before its actual usage. In this stage, as validation is already done, the program will proceed rest of the execution with erroneous value. Any compiler optimization to reduce register lifetime will be helpful in this scenario.

AR20 demonstrates a comparable level of protection rate to SWIFT-R by exhibiting a 95.67% protection rate with 2.23% of SDCs and 1.63% of Segfaults. Protection rate decreases when acceptable range broadens: AR20 (95.67%), AR50 (94.51%), AR80 (93.42%), AR100 (92.52%). The differences in protection rate are mainly originated by SDCs because we protect address calculation of memory instruction with the conventional strategy: AR20 (2.23% SDCs), AR50 (3.37% SDCs), AR80 (4.30% SDCs), AR100 (5.29% SDCs). Throughout the experiment, the occurrence for Core dump and Hang in every protection scheme, including UNSAFE and SWIFT-R, is measured less than 0.3%.

Figure 9b presents the statistics of false negative. As expected, its occurrence increases with widened acceptable range: AR20 (1.80%), AR50 (3.12%), AR80 (3.74%), AR100 (5.04%). As fuzzy validation is applied to data validation, false negatives mostly produce SDCs. Interestingly, due to *YOLOv2*'s program characteristics, false negatives are generally benign in this application. After extensive computations through multiple layers, only a label with the highest probability for each detected object is produced as the output. Naturally, small errors that eschewed fuzzy validations tend to be logically masked in later part of the execution.

7.3 The Rationality of Acceptable Range

Before the deployment of RSkip, the acceptable range of fuzzy validation should be determined with consideration of both protection rate and performance. On average, the representative conventional scheme, SWIFT-R, can provide 97.24% of protection rate with 2.33× slowdown. As for widening acceptable range, RSkip achieves significant performance benefits in exchange of certain loss for the protection rate : AR20 (95.67% of protection rate with 1.42× slowdown), AR50 (94.51% of protection rate with 1.33× slowdown), AR80 (93.42% of protection rate with 1.30× slowdown), AR100 (92.52% of protection rate with 1.27× slowdown). With AR20, RSkip can significantly reduce protection overhead while presenting comparable level with the conventional scheme. Also, RSkip can reduce the overhead further to 1.27× with the same amount of loss (5 percentage point) that previous works [16, 17] leveraged.

8 Related Works

Hardware techniques insert additional hardware module to protect their execution. For example, Austin proposed DIVA [4, 43] checker which is a small core designed to validate the computation on the fly. Despite their expensive cost, hardware techniques have been adopted in real systems for the high fidelity [15, 38, 45]. Additionally, Racunas et al. [26] proposed a perturbation-based screening hardware technique considering an erratic value as a possible fault. Legitimate sets of values are defined to detect inconsistency in values.

After Saxena et al. [36] noticed the SMT redundancy, many techniques were proposed to reduce hardware cost [14, 23, 39]. To improve performance of thread-level approach, DAFT [46] exploits speculation to minimize inter-thread communications for memory operations. Yet, utilizing redundant threads generally suffers from high energy consumption [19] due to an increased number of dynamic instructions for redundant threads and validation.

Additionally, instruction duplication based protection was first introduced by Oh et al. [25] where all instructions including memory operations were duplicated and validated. Reis et al. proposed SWIFT [31] to optimize instruction duplication scheme by removing unnecessary memory redundancies based on the assumption of ECC. As SWIFT did not claim any recovery method, Reis et al. expanded SWIFT further with TMR-based instruction level recovery technique [30]. In general, fault detection and fault recovery mechanism can be investigated independently [17, 31]. Encore [13] and checkpoint-based methods [12, 42] are proposed as the independent recovery scheme. At wrong predictions, our prediction-based protection scheme executes re-computation for the exact validation. In this case, the mechanism generating re-computation for the redundant copy resembles a form of recovery mechanism. Also, unlike previous works [11, 17, 30, 31] that trigger recovery routine only on the actual fault detection, our approach additionally triggers recovery mechanism on mispredictions. To enhance performance of both detection and recovery mechanisms in RSkip, the integration of advanced recovery mechanisms [12, 13, 42] can be studied.

To reduce the protection cost, Khudia et al. [17] leveraged certain loss in protection rate by protecting only critical variables. Similarly, Venkatagiri et al. proposed Approxilyzer [41], a framework that suggests protecting critical dynamic instructions by quantifying quality impact of a single bit flip to avoid excessive protection cost. In contrast to our work, they reduce protection cost by narrowing down protection targets. By incorporating this technique, a better performance can be achieved.

9 Conclusion

The unique properties of a transient fault force protection strategies to be fast and cost-efficient. RSkip demonstrates that the performance of the software-only protection strategy can be significantly improved with a newly proposed prediction-based protection scheme. By predicting the value of computation with approximation techniques, the expensive re-computation can be bypassed. As a result, 81.10% of re-computation can be skipped on average with a remarkable performance improvement. While conventional technique suffers from 2.33x slowdown compared to the unreliable execution, RSkip only suffers 1.27x slowdown with a 5% loss in protection rate.

Acknowledgement

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), under Award Number DE-SC0014134.

References

[1] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. 2009. *Using code perforation to improve performance, reduce energy consumption, and respond to failures.* Technical Report. Technical report, MIT.

[2] Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.* 54, 7 (2005), 922–927.

[3] Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2012. Dynamic tolerance region computing for multimedia. *IEEE Trans. Comput.* 61, 5 (2012), 650–665.

[4] Todd M Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on.* IEEE, 196–207.

[5] Woongki Baek and Trishul M Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, Vol. 45. ACM, 198–209.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* ACM, 72–81.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[8] Shekhar Borkar et al. 2004. Microarchitecture and design challenges for gigascale integration. In *MICRO*, Vol. 37. 3–3.

[9] Greg Bronevetsky, B de Supinski, and Martin Schulz. 2009. *A foundation for the accurate prediction of the soft error vulnerability of scientific applications.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* Ieee, 44–54.

[11] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: A Compiler technique for near Zero Silent data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference.* ACM, 48.

[12] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 385–396.

[13] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A Mahlke, and David I August. 2011. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 398–409.

[14] Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. 2003. Transient-fault recovery for chip multiprocessors. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on.* IEEE, 98–109.

[15] Robert W Horst, Richard L Harris, and Robert L Jardine. 1990. Multiple instruction issue in the NonStop Cyclone processor. In *ACM SIGARCH Computer Architecture News*, Vol. 18. ACM, 216–226.

[16] Daya Shanker Khudia and Scott Mahlke. 2013. Low cost control flow protection using abstract control signatures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 3–12.

[17] Daya Shanker Khudia and Scott Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on.* IEEE, 319–330.

[18] Shikai Li, Sunghyun Park, and Scott Mahlke. 2018. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *Proceedings of the 2018 International Conference on Supercomputing.* ACM, 341–351.

[19] Niti Madan and Rajeev Balasubramonian. 2007. Power efficient approaches to redundant multithreading. *IEEE Transactions on Parallel and Distributed Systems* 18, 8 (2007).

[20] Panagiotis D Michailidis and Konstantinos G Margaritis. 2013. Accelerating kernel density estimation on the GPU using the CUDA framework. *Applied Mathematical Sciences* 7, 30 (2013), 1447–1476.

[21] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2012. *Introduction to linear regression analysis.* Vol. 821. John Wiley & Sons.

[22] Shubu Mukherjee. 2011. *Architecture design for soft errors.* Morgan Kaufmann.

[23] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on.* IEEE, 99–110.

[24] Shubhendu S Mukherjee, Christopher T Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. 2003. Measuring architectural vulnerability factors. *IEEE Micro* 23, 6 (2003), 70–75.

[25] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002), 63–75.

[26] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S Mukherjee. 2007. Perturbation-based fault screening. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on.* IEEE, 169–180.

[27] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[28] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *arXiv preprint arXiv:1612.08242* (2016).

[29] Steven K Reinhardt and Shubhendu S Mukherjee. 2000. *Transient fault detection via simultaneous multithreading.* Vol. 28. ACM.

[30] George A Reis, Jonathan Chang, and David I August. 2007. Automatic instruction-level software-only recovery. *IEEE micro* 27, 1 (2007).

[31] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization.* IEEE Computer Society, 243–254.

[32] Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. 2008. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on.* IEEE, 70–79.

[33] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 35–50.

[34] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 13–24.

[35] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The Bunker Cache for spatio-value approximation. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 1–12.

[36] Nirmal R Saxena and Edward J McCluskey. 1998. Dependable adaptive computing systems-the roar project. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, Vol. 3. IEEE, 2172–2177.

[37] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. 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.* ACM, 124–134.

[38] Timothy J Slegel, Robert MIII Averill, Mark A Check, Bruce C Giamei, Barry W Krumm, Christopher A Krygowski, Wen H Li, John S Liptay, John D MacDougall, Thomas J McPherson, et al. 1999. IBM's S/390 G5 microprocessor design. *IEEE micro* 19, 2 (1999), 12–23.

[39] Jared C Smolens, Jangwoo Kim, James C Hoe, and Babak Falsafi. 2004. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 257–268.

[40] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[41] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 1–14.

[42] Nicholas J Wang and Sanjay J Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 188–201.

[43] Chris Weaver and Todd Austin. 2001. A fault tolerant approach to microprocessor design. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 411–420.

[44] Vincent M Weaver, Dan Terpstra, Heike McCraw, Matt Johnson, Kiran Kasichayanula, James Ralph, John Nelson, Phil Mucci, Tushar Mohan, and Shirley Moore. 2013. Papi 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 124–125.

[45] Ying C Yeh. 1996. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, Vol. 1. IEEE, 293–307.

[46] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. 2012. DAFT: decoupled acyclic fault tolerance. *International Journal of Parallel Programming* 40, 1 (2012), 118–140.

[47] James F Ziegler and Helmut Puchner. 2004. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress.