Harnessing Soft Computations for Low-budget Fault Tolerance

Daya Shanker Khudia and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI
{*dskhudia, mahlke*}@*umich.edu*

***Abstract*—A growing number of applications from various domains such as multimedia, machine learning and computer vision are inherently fault tolerant. However, for these soft workloads, not all computations are fault tolerant (e.g., a loop trip count). In this paper, we propose a compiler-based approach that takes advantage of soft computations inherent in the aforementioned class of workloads to bring down the cost of software-only transient fault detection. The technique works by identifying a small subset of critical variables that are necessary for correct macro-operation of the program. Traditional duplication and comparison is used to protect these variables. For the remaining variables and temporaries that only affect the micro-operation of the program, strategic expected value checks are inserted into the code. Intuitively, a computation-chain result near the expected value is either correct or close enough to the correct result so that it does not matter for non-critical variables. Overall, the proposed solution has, on average, only 19.5% performance overhead and reduces the number of silent data corruptions from 15% down to 7.3% and user-visible silent data corruptions from 3.4% down to 1.2% in comparison to an unmodified application. This unacceptable silent data corruption rate is even lower than a traditional full duplication scheme that has 57% overhead.**

***Keywords*-Soft Errors; Compiler Analysis;**

I. Introduction

An increasing number of both current and emerging workloads from domains such as multimedia, machine learning and computer vision either compute on approximate data and/or produce results that have subjective interpretations, *i.e.* the quality of the output is subjectively judged by a human. Such applications can inherently tolerate more faults while still producing user acceptable outputs. User acceptable outputs are the program outputs where either the user can not differentiate between an output in presence of a fault or the output is useful even in presence of fault. Multimedia computations such as encoding/decoding of audio, images and video are examples of such applications. Such computations are referred to as soft or imprecise computations [1] in the literature. Also, other applications from domains such as machine learning and computer vision use probabilistic algorithms that are inherently tolerant to a certain degree of faults.

The focus of our work is on the faults caused by soft errors. **Soft errors**, also referred to as Single Event Upsets (SEUs) or **transient faults**, are caused by high energy particle strikes. Soft errors can also be caused by circuit crosstalk or random noise. The silicon-chip technology is becoming more susceptible to soft errors with each new generation due to decreasing transistor sizes and increasing transistor density. Soft Error Rate (SER) for the logic on chip is steadily rising with technology scaling [2]. SER is the rate at which a component encounters soft errors and SER scales with number of transistors and level of integration [3]. Soft computing workloads have high levels of inherent fault tolerance. For such workloads, fault detection efforts can be directed only to the parts of the program that when perturbed produce user unacceptable outputs. As a result, there is an opportunity to reduce the overhead of fault protection for these applications. In this work, we analyze and identify the nature of faults that cause unacceptable outputs and propose an efficient software-only fault detection scheme that exploits soft computations.

The inherent fault tolerant nature of soft computing applications raises an important question: *Do these applications require any fault protection at all?* The answer to this question is ”yes, they do” because not all computations in soft computing applications are fault tolerant. As identified by the works in the field of approximate computing [4], [5], a program has certain computations that can be approximate for user acceptable outputs, while the computation of other parts of the program needs to be precise. For example, correctness of a variable that holds the number of frames of video to be decoded is more important for user acceptable output than the computation of a single pixel in a frame. To differentiate errors causing the user acceptable outputs from to the ones causing unacceptable outputs, we refine the definition of silent data corruption to **Unacceptable Silent Data Corruptions** (USDCs). USDCs are the incorrect program outputs in presence of a fault that are below an acceptable quality but the program completes execution without terminating prematurely and behaving abnormally.

Our solution takes advantage of *not-so-strict* requirement on program output correctness and protects only the critical parts of the computation. To this end, we analyze the nature of soft computations and propose a compiler-based software-only approach for identifying USDC-causing variables automatically and inserting relevant detection code. Our approach does not require any program annotations and works by identifying critical variables that, if corrupted, affect the program output significantly as a single corruption

either affects many computations or has repeated impact on computation. Variables that carry a state across iterations in a loop are examples of such critical variables. Computation of critical variables is protected using traditional replication, duplicating their producer chain and inserting a check [6]. Expected value checks are inserted on other variables to make sure that they stay in a compact range obtained by profiling. We hypothesize that a deviation outside this range is unlikely to happen in program execution under normal conditions. Any deviations within the checking range is unlikely to cause a USDC. Hence, expected value checks represent checking substantial abnormal behavior of a program while allowing insignificant corruptions. In this manner, soft checks are performed on soft computation because they are low overhead, while hard checks are sparingly used on critical variables.

The major contributions of this work are as follows:

- A fully automated compiler analysis and transformation method that partitions computations among three categories: to be protected by traditional duplication, to be protected by soft value checks or not to be protected. This method also judiciously performs selective duplication and inserts value checks. Our technique does not require any program annotations.
- We analyzed soft computing benchmarks from various domains such as multimedia, machine learning and computer vision to identify the nature of computations and to develop compiler heuristics. We also implemented fidelity metrics to measure the objective quality of the outputs.
- Fault injection experiments are performed to evaluate the efficacy of the proposed scheme. We show that, on average, at 19.5% performance overhead, SDCs can be reduced from 15% down to 7.3% and USDCs from 3.4% down to 1.2% in comparison to an unmodified application. This unacceptable silent data corruption rate is even lower than a traditional full duplication scheme that has 57% overhead.

II. Motivation

A. Soft Computations

Soft computations can tolerate relatively higher numbers of errors than other applications that require their results to be numerically-precise. Soft computing has been previously exploited in trading off accuracy for energy efficiency or execution time [4], [5]. In this paper, we propose to exploit such computations for trading off the cost of providing reliability with the accuracy of results. However, all parts of these error tolerant applications are not equally error tolerant. For example, errors in loop variables might cause a significant portion of the output to be corrupted. The computation of such variables needs to be precise.

In order to define the level of acceptable degradation, we need to evaluate whether the output of an application is acceptable to the end user. Naturally, the tolerable amount of degradation is application dependent and different quality metrics are required for different applications. For example, an objective metric for the acceptable quality of a decoded image is to have Peak Signal to Noise Ratio (PSNR) above a certain threshold. Higher PSNR implies a better quality image. Similarly, the output of a classification algorithm (machine learning application) can be acceptable if the number of correctly classified test data in presence of a fault does not significantly differ from the classification accuracy in the absence of the fault. The type of quality measure metric used for different applications and thresholds for them to be of accepted quality are provided later in Section IV-B.

(a) Decoded image in a fault-free environment (b) Decoded image of acceptable quality in presence of a fault (c) Decoded image of unacceptable quality in presence of a fault

Figure 1: Difference between decoding (part (a)) of an image in a fault-free environment and decoding in presence of faults (part (b) and (c)). Though the decoded image in part (b) does not numerically match with fault free decoding, the difference is not perceptible. The distortions in part (c) are perceptible (top-right corner) and thus the output is unacceptable.

In Figure 1, we demonstrate how the faults might affect the output of an application. We injected faults into various runs of a *jpeg* image decoder and analyzed the outputs. The experimental setup for injecting faults is described in Section IV. Figure 1 shows an example of a decoded image under three scenarios. Figure 1(a) is the decoded image when no fault was injected in the application run. Figure 1(b) is the decoded image when a fault was injected and the output is numerically incorrect but the difference is not perceptible. Figure 1(c) shows the unacceptable output in presence of a fault. The top-right portion of Figure 1(c) is significantly distorted due to incorrect pixel values. The pixels in both Figure 1(b) and Figure 1(c) do not numerically match with the ones in Figure 1(a). However, the primary difference between these two figures is that in Figure 1(b) only few of pixels are incorrectly computed thus causing an imperceptible difference while in Figure 1(c), a large slice of pixels are incorrectly calculated causing a perceptible change. We further analyzed the propagation of the faults in each of these cases. The fault in Figure 1(b) only corrupts the output of an addition by a small amount while calculating inverse discreet cosine transform, hence causes only small output disturbance. However, the fault in Figure 1(c) causes error

in decoding Huffman-compressed coefficients for a block of data, hence corrupting a lot more data. This demonstrates that for soft computing applications it is critical to protect the computations that affect a large amount of output.

B. *Silent Data Corruptions*

The fault tolerance research community only considers a cycle-by-cycle match of architectural state as the correct execution of a program. This strict notion of program correctness is called Architecturally Correct Execution (ACE) [7]) and is used in many hardware based reliability solutions. However, Li et al. [8] showed that 17.6% of the multimedia and AI application runs produced correct results even though they had architecturally incorrect states. Feng et al. [6] believe that user-visible output corruptions are what truly matters, and Khudia et al. [9] also leverage this idea of application level correctness. A program is said to have an SDC, if in presence of a fault, the program completes execution without terminating prematurely and behaving abnormally but the output of the program is incorrect. These are most harmful type of faults because they silently corrupt the output of the program while the user thinks that program worked correctly. Hence, a number of previous works [10], [6] have analyzed SDCs and focused on reducing them.

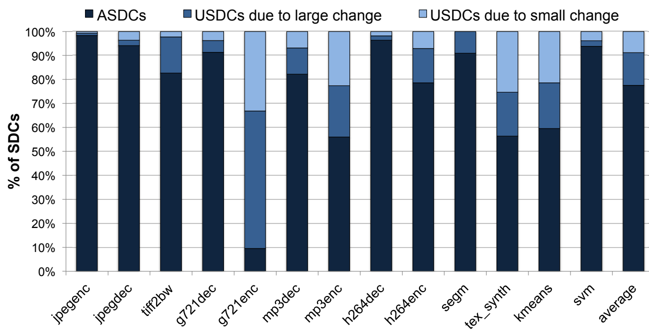

Figure 2: SDCs are divided into acceptable SDCs and unacceptable SDCs. Unacceptable SDCs are further divided into the ones due to large and small instruction output value changes. Soft checks using expected values can detect unacceptable SDCs (up to 14% of total SDCs) due to large instruction output value change.

Traditionally, SDC-free execution is considered as a criterion for correctness. However, for soft computations the program can be assumed to have correctly executed even if it generates numerically incorrect but high fidelity outputs. This notion of having acceptable corruption is not applicable to all types of applications, e.g., most of the SPEC CPU benchmarks, but is applicable to soft computation benchmarks. For our work, we divide SDCs further into two categories: Acceptable Silent Data Corruptions (ASDCs) and Unacceptable Silent Data Corruptions (USDCs). ASDCs are the SDCs that are admissible to the user due to the negligible differences compared to fault free execution. However, USDCs are the SDCs that change the output significantly such that it is not acceptable to the user.

We performed fault injection experiments on unmodified soft computing benchmarks to quantify the USDCs caused by faults that force large disturbance on the generated values by instructions. The results of this experiment are presented in Figure 2. The experimental setup and description of these benchmarks are presented later in Section IV. The Y-axis in the graph plots total SDCs caused in the fault injection experiments. Each column is divided into ASDCs and USDCs. USDCs are further divided into the SDCs that were due to a large and small value changes in the corrupted instructions. On average, 77% of the SDCs result in ASDCs and 14% in USDCs with large value changes. ASDCs are the errors that result in user acceptable outputs and therefore nothing needs to be done for these. USDCs that are caused by large output value change of a computation can be detected by having expected value checks. Expected value checks, obtained by profiling (Section III-C1), make sure that the output does not deviate outside a compact range. Protecting against 23% USDCs might not seem significant but one must view these errors in the proper context as USDCs are the worst of the worst.

III. Solution: Analysis and Design

A. *Overview*

We analyze a number of benchmarks to find out the most vulnerable computations in soft applications to develop our compiler heuristics. These analyses involve fault injections and then investigating fault propagation. The outcome of the program is correlated back to the fault injection variable. Once these patterns are identified, we make compiler heuristics to insert checking code in the application. In our experiments, we found that the developed heuristics work well across a wide range of soft applications.

```
1          ...
2 for(crc = init; len >= 32; len -= 32){
3   register unsigned long data;
4   data = mad_bit_read(&bitptr, 32);
5          ...
6   tableVal = crc_table[(data >> 24) ^ ...];
7   crc = (crc << 8) ^ tableVal;
8          ...
9 }
10         ...
```

Figure 3: The code snippet from *mp3dec (mad)* [11] benchmark. The variables that are dependent on their own values in the previous iterations are underlined. A corruption in such variables is more likely to result in unacceptable outputs.

To show frequently occurring computations in soft computing applications, we show a code snippet from *mp3dec (mad)* [11] benchmark in Figure 3. In our experiments with various benchmarks, we have noticed that a corruption in the variables that carry state across loop iterations is more

```
1           ...
2 crcD = init;
3 for(crc = init; len >= 32; len -= 32){
4   register unsigned long data;
5   data = mad_bit_read(&bitptr, 32);
6           ...
7   tableVal = crc_table[(data >> 24) ^ ...];
8   crc = (crc << 8) ^ tableVal;
9   crcD = (crcD << 8) ^ tableVal;
10  if(crc != crcD)
11    recover_and_continue_execution();
12          ...
13 }
14          ...
```

Figure 4: The code snippet from Figure 3 with *crc* variable duplicated. For the sake of brevity, the duplication of other state variables (those shown in Figure 3) is not shown in this figure. Variables postfixed with *D* are duplicated variables.

likely to result in USDCs. We define such variables as **state variables** and these variables are underlined in this figure. State variables include loop iteration variables. Intuitively protecting state variables makes sense as state variables have a *snowball effect* on the subsequent computations, because the error not only affects the current iteration but it also propagates to future iterations. Protecting such variables is critical to minimize the user unacceptable outputs because errors in these variables are likely to cause significant changes in the output of a program. Loop index variables are also state variables and an error in loop index variables have the potential to change the output significantly by increasing or decreasing the number of iterations executed.

We protect state variables by duplicating the producer chains of such variables. Producer chain of a variable can be obtained by the recursive traversal of its use-def chain. The effect of duplicating the producer chain of one such variable *crc* is shown in Figure 4. Line 9 in Figure 4 is the duplicated line and variables postfixed with *D* are the duplicated variables. For the purpose of exposition, we deliberately show duplication of only a single variable in this example. A more detailed and complete example of duplication is presented later in this section.

In general, some variables and instructions generate a value or a range of values frequently [12]. Generation of such values is more common in soft computations due to the repetition of same calculation on different inputs.

A check for these frequent values or a range of values produced by an instruction can help protect against corruption. A range check is inserted for such variables and the range is obtained by profiling, as explained later in Section III-C. Figure 5 shows a value-range check inserted for a variable on line 7. This is assuming that the variable *tableVal* lies between V1 and V2. If the duplication were to be performed for *tableVal*, its input *data* and *data*'s producer chain would also need to be duplicated. Thus the value checks help to save on cost of duplication. Again for the purpose of

```
1           ...
2 for(crc = init; len >= 32; len -= 32){
3   register unsigned long data;
4   data = mad_bit_read(&bitptr, 32);
5           ...
6   tableVal = crc_table[(data >> 24) ^ ...];
7   if(tableVal < V1 || tableVal > V2)
8     recover_and_continue_execution();
9   crc = (crc << 8) ^ tableVal;
10          ...
11 }
12          ...
```

Figure 5: The code snippet from Figure 3 with expected value check inserted on variable *tableVal*. Assume that the value generated lie within the range [V1, V2] (Obtained by profiling). This is a simple example of inserting value checks and more detailed examples are shown later in Section III-C.

exposition, Figure 5 only shows a simple example.

If an instruction generates the same value frequently then this value can be used to check the output of that instruction at certain opportune points in an application. The use of frequently generated values for soft checks is a novel idea but frequently generated values by an instruction has previously been used in various optimizations [12]. For example, if a multiply operation generates the same invariant value frequently, then the multiply operation can be optimized away with a check inserted for the correct value. Racunas et al. [13] also make use of certain consistent bounds on intermediate data in their hardware-based scheme. The intuition behind such value-range checks is that if the instructions produce values between previously seen ranges (in the profile data) the output is not significantly affected and is expected to be acceptable. These checks are soft checks in the sense that they check the expected output values of instructions.

Overall, the foundation of our work is based on the following two observations:

1) First, if the program variables in the main loops of applications that have state across iterations are corrupted, they are more likely to result in unacceptable output. Protecting these variables is critical for reducing USDCs.
2) Second, many soft computing benchmarks use the same calculations on different inputs repeatedly such that generated values are in a range. If in presence of an error, the value generated is within this range, it is probabilistically unlikely to have a USDC in such a case. An expected value check on such instructions is inserted to protect against soft errors.

B. Recomputing State Variables

State variables are a critical part of an application and corruption in these variables propagates to subsequent iterations of the loop. They are protected by duplicating their producer chain and then inserting a comparison between the original

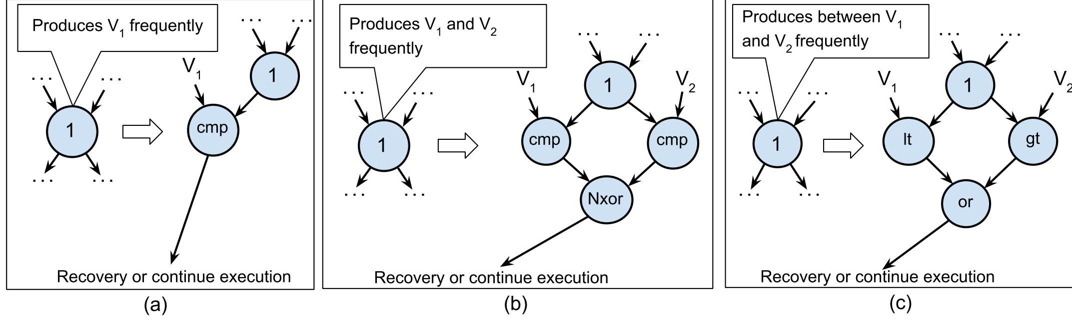

Figure 6: Depending on the generated values, one of the three different types of value checks can be inserted. Part (a) shows a single value check inserted if a single value is frequently generated by an instruction. If two values are most frequently generated, the check in part (b) is inserted. However, if the values generated lie in a range, a range check as shown in part (c) is inserted.

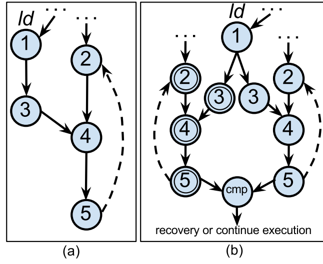

Figure 7: Instruction duplication in a single thread of execution. Instructions marked with double circle are duplicated instructions. The instruction marked with *ld* is a load instruction. We do not duplicate loads to save on memory traffic.

producer chain and the duplicated producer chain. The technique to identify state variables is described in Section IV-A. Figure 7 shows an example of such a duplication process in the form of a data flow graph. Each circle represents an instruction (or destination variable of that instruction). The solid arrows are data flow edges and dashed arrows represent inter-iteration loop dependences. The instructions marked with *ld* is a load instruction. The instructions marked with double circle are duplicated instructions. The state variable in this figure is variable 5. The producer chain of instruction 5 is duplicated as shown in Figure 7(b). To save on memory traffic, the producer chain is terminated whenever a *load* instruction is encountered. The reason for this is that a fault in data flow input for load (address operand) is more likely to result in a symptom such as out-of-bound access. Such symptoms can be used as an indication of soft error [6], [14] and a recovery can be triggered.

C. Expected Value Checks

In soft computing benchmarks, same calculation on different inputs is performed repeatedly. Moreover, in general some instructions produce the same value almost all the times [12]. To cover the values produced by an instruction, we devise three different types of value checks as shown in Figure 6. Figure 6(a) shows the data flow graph before and after the value checks are inserted. Instruction 1 produces the value V_1 frequently so a check with V_1 is inserted. Similarly, Figure 6(b) shows the code before and after value checks if the instruction generated two values V_1 and V_2 most frequently. Finally, Figure 6(c) shows the data flow graph before and after a range check is inserted on an instruction that produces values in a range $[V_1, V_2]$. To optimize the number of value checks, we came up with two optimizations for long producer chains.

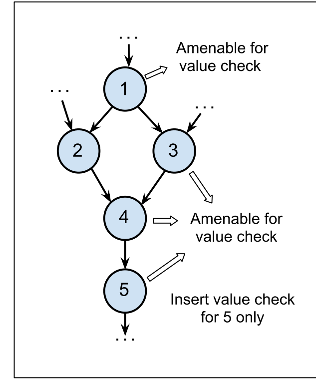

Figure 8: Optimization 1 for long producer chains: this figure shows an example of a case where multiple instructions in the producer chain of an instruction are amenable for value checks. In order to minimize on number of checks, value check should only be inserted for an instruction lower in the producer chain.

Optimization 1: A naive insertion of value checks on all the instructions that produce values amenable for one of the checks in Figure 6 might lead to a prohibitively large number of checks. Hence, to reduce the number of checks, we insert value checks deeper in the producer chain. Figure 8 shows an example of such an optimization. If the values produced by instruction 1, 3, 4 and 5 are amenable for value checks, a value check is only inserted on the value produced by instruction 5.

Optimization 2: While duplicating instructions, if in a long producer chain, an instruction produces a value amenable to checks, the duplication is terminated and a value check is inserted. An example of this is shown in Figure 9. In this example, instruction 4 produces value(s) or a range. In our duplication framework, if such a situation is encountered

the recursive duplication of producers is terminated and one of the value checks is inserted instead.

1) Value Profiling:: The frequent values or the range of values produced by an instruction are obtained using value profiling. In general, during the profile run, collecting all the values produced by an instruction has very high overhead. An optimization to this is to maintain a fixed set of most frequently produced values by each instruction. Since we also need a range of values produced by an instruction, we have modified this traditional value profile. Essentially, we require a histogram with bins as values produced corresponding to each instruction. However, the future values are unknown, so deciding the histogram bin size before running the program is not possible. We have adopted a modified version of the On-line histogram algorithm [15] for this purpose. Our adopted version of the algorithm is shown in Algorithm 1.

The algorithm takes a histogram h of size B as an input. The number of bins B are pre-decided and are set to 5 in our experiments. Initially the input histogram to this algorithm can be empty. This histogram is maintained for every value generating instruction in the program during profiling phase (a one time off-line process). ([lb_1, rb_1], m_1) is a bin frequency pair and m_1 represents the number of values generated by a particular instruction between and including lower bound of the bin (lb_1) and upper bound of the bin (rb_1).

Once we have the bin-frequency pair for all the value generating instructions in an application, the next step is to obtain a tight range of lower and upper bound where most of the values generated by an instruction are concentrated. This information is calculated and used while inserting the value checks in the application source code. This is obtained by a greedy algorithm that works by picking a bin that has highest frequency and extends this bin towards left or right while the range size lies within a threshold. This algorithm is shown in Algorithm 2.

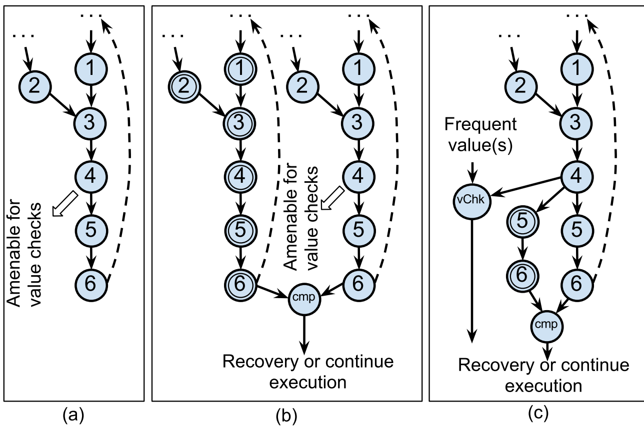

Figure 9: Optimization 2 for long producer chains: if an instruction amenable to value check is encountered in producer chain, the duplication of producer chain of critical variables is terminated at that point and a value check (*vChk*) is inserted as shown.

Input: A histogram h = ([lb_1, rb_1], m_1), ... ([lb_B, rb_B], m_B), a value v
Output: A histogram with B bins

1 **if** $v \in$ *[lb_i, rb_i] for some i* **then**
2 $\quad m_i$ = m_i + 1
3 **end**
4 **else**
5 $\quad$ Add $[v, v, 1]$ to the histogram h. Histogram h can now potentially have B+1 bins;
6 $\quad$ Sort the bins. Denote the sorted bins by ([lp_1, rp_2], m_1), ... ([lp_{B+1}, rp_{B+1}], m_{B+1});
7 $\quad$ Find a bin [lp_i, rp_i] that minimizes lp_{i+1} - rp_i;
8 $\quad$ Replace the bins ([lp_i, rp_i], m_i), ([lp_{i+1}, rp_{i+1}], m_{i+1}) by the bin
9 $\quad$ ([lp_i, rp_{i+1}], m_i + m_{i+1});
10 **end**

Algorithm 1: Algorithm for obtaining histogram of the values produced by an instruction.

Input: A histogram h = ([lb_1, rb_1], m_1), ... ([lb_B, rb_B], m_B) with sorted bins, a threshold on range R_{thr}
Output: A frequent range ([lp, rp], m)

1 Pick a bin ([lb_i, rb_i], m_i) such that m_i = max(m_1 ... m_B);
2 initialize retBin = ([lb_{ret}, rb_{ret}], m_{ret}) with ([lb_i, rb_i], m_i);
3 Denote the bin left to retBin by leftBin and the one to the right by rightBin;
4 leftBin = ([lb_{left}, rb_{left}], m_{left}) and rightBin = ([lb_{right}, rb_{right}], m_{right});
5 **while** *(rb_{ret} - lb_{ret} ¡ R_{thr}) and still unconsidered bins* **do**
6 $\quad$ **if** $m_{left} \geq m_{right}$ **then**
7 $\quad\quad$ retBin = ([lb_{left}, rb_{ret}], m_{left} + m_{ret});
8 $\quad\quad$ leftBin = next leftBin;
9 $\quad$ **end**
10 $\quad$ **else**
11 $\quad\quad$ retBin = ([lb_{ret}, rb_{right}], m_{ret} + m_{right});
12 $\quad\quad$ rightBin = next rightBin;
13 $\quad$ **end**
14 **end**
15 return retBin;

Algorithm 2: A greedy algorithm for obtaining compact range on the values produced by an instruction.

An important point to note here is that value profiling is an offline process (needs to be done once per benchmark) and this overhead does not directly impact the performance overhead of our technique. The frequent values or frequent range of values are obtained by profiling the program on representative inputs. An application instrumented with

Table I: The benchmarks and their acceptable quality metrics.

Benchmark (Suite)	Description (Category)	Inputs		Fidelity Measure (Threshold)
		train	test	
jpegenc and jpegdec (mediabench [16])	A JPEG image encoder/decoder (image)	600x450 image	256x256 image	Peak Signal to Noise Ratio (PSNR) (30 dB)
tiff2bw (mibench [11])	A tiff format to BW converter (image)	3069x3100 image	1520x1496 image	PSNR (30 dB)
segm (SDVBS [17])	Image segmentation (Computer vision)	176x132 image	44x33 image	Segment matrix mismatch (10%)
tex_synth (SDVBS [17])	Texture synthesis (Computer vision)	20x20 image	16x16 image	Output matrix mismatch (10%)
g721enc and g721dec (mediabench [16])	audio encoding and decoding (audio)	9.6MB audio	.3MB audio	Segmental SNR (80 dB)
mp3enc and mp3dec (mibench [11])	mp3 encoding and decoding (audio)	2.6MB audio	.18MB audio	PSNR (30 dB)
h264enc and h264dec (mediabench II [18])	h264 video encoding and decoding (video)	5.3MB video	.23MB video	PSNR (30 dB)
kmeans (in-house)	Clustering algorithm (Machine learning)	100x9 samples	100x18 samples	Cluster assignment mismatch (10%)
svm (svmlight [19])	Support vector machine (Machine learning)	2000 train examples	2000 test examples	Classification error (10%)

expected value checks might generate value check failures at runtime even if there are no errors (false positives). However, this is not a correctness issue and could only lead to unwanted recovery initiations. If a check fails, recovery from the check should be executed once and if the same check fails again after recovery further recovery should not be executed for that check. False positives rate is analyzed in Section V.

IV. Experimental Setup

We have evaluated our work by Statistical Fault Injections (SFIs) into a microarchitectural model of a modern microprocessor. This same method is used by previous works [6], [9], [7] to evaluate reliability solutions. SFI is performed by introducing bit flips randomized in both time and space.

A. Source Code Transformations

We use the LLVM [20] compiler infrastructure to insert duplication code and expected value checks into the application's source code. At first, application source code is converted into LLVM's internal representation called LLVM IR (Intermediate Representation). Our solution is implemented as a pass over LLVM IR. Value profiling is implemented as a separate pass. The IR is instrumented to collect value profiling information. Our duplication pass uses information from other analysis passes such as value profiling to produce bitcode with duplicated instructions and value checks. The LLVM code generation framework is then used to generate ARM binaries from the modified bitcode. **Identifying State Variables:** LLVM IR is in Static Single Assignment (SSA) form. At IR level, the state variables can simply be identified by looking the *phi* nodes in loop headers. A *phi* node merges all the incoming versions of the variable to create a new name for it. State variables have two incoming definitions—one from outside loop definition and the other from inside loop updates—at loop headers and are represented by *phi* nodes in loop headers.

B. Benchmarks and Fidelity Measures

We have collected a variety of benchmarks (a total of 13) that represent soft computations from various domains and at least two benchmarks from each of the following five categories: image, audio and video processing; computer vision; machine learning. These benchmarks represent a good mix of soft computations. A brief description of these benchmarks along with their source benchmark suite is given in Table I. Different inputs are used for profiling and running the benchmarks. These profiling and test inputs are given in column 3 of the table. Column 4 in the table shows the fidelity metric used to evaluate the quality of the produced outputs. This is an application dependent metric and different metrics are used for different benchmarks. Column 4 also shows the threshold used for acceptable quality results. Higher PSNR represents a better quality image and video. We chose 30 dB as threshold for PSNR and 80 dB for segmental SNR for acceptable quality. Similar threshold values are used by Thomas et al. [21]. For machine learning and computer vision benchmarks, outputs with more than 10% deviations are not considered acceptable. All these benchmarks are compiled with their suggested compiler options. Figure 10 shows the total number of state variables, duplicated instructions and inserted value checks as a fraction of the total static IR instructions. At most, only 11.4% of the static IR instructions are duplicated and only 8.3% of total static IR instructions have expected value checks on them.

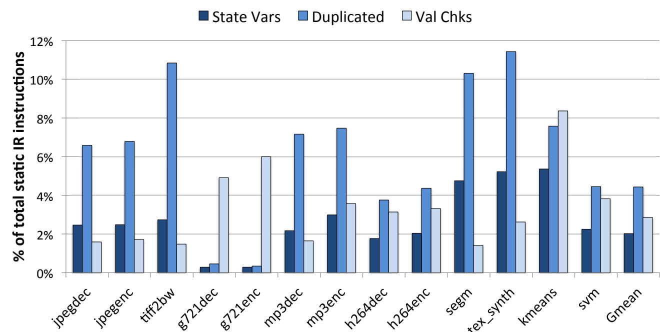

Figure 10: shows the total number of state variables, duplicated instructions and inserted value checks as a fraction of the total static IR instructions. The static code duplication and expected value checks are not more than 12% of the total static IR instructions.

C. Fault Model and Injection Experiments

The proposed approach is evaluated by injecting a number of faults in each application run. The traditional single bit-flip [22], [8], [21], [6] in the processor state is used to model transient faults. At a random cycle during the program execution, a register is randomly selected first and then a randomly selected bit in that register is flipped. Wang et al. [23] showed that, on aggregate, as much as 70% of the total failures due to faults in pipeline structures such as register file, register alias tables, register free lists, instruction input and output operands etc. results in register file inconsistencies. Thus, the register file is an enticing target for fault injections and similar to previous works [6], [9], we evaluated our work by injecting faults into the register file. Please note that protecting register file by ECC would be able to cover faults occurring in register file itself but not the faults that occur in other hardware structures and then propagate to register file. Overall, our proposed technique is capable of handling faults in other microarchitectural units that affect the program. A fault in a register can also affect the data dependent control flow. Our solution protects against such faults either by state variables duplication or by value checks. However, it does not provide protection against faults that affect branch targets. For protecting against branch target faults, a previously proposed [24] signature-based low-cost solution can be used in conjunction with our proposed approach.

Table II: GEM5 Simulator parameters (models an ARMv7-a profile of the ARM architecture).

Processor core @ 2GHz	
Simulation configuration	out-of-order core
Simulation mode	Syscall emulation
Physical integer register file size	256 entries
Reorder Buffer Size	192 entries
Issue width	2
Memory	
L1-D cache	64KB, 2-way
L1-I cache	32KB, 2-way
DTLB/ITLB	64 entries (each)

We used the GEM5 [25] simulator to simulate the workloads and implemented fault injection infrastructure in this simulator. The simulator was run in ARM syscall emulation mode and modeled the ARMv7-a profile of ARM architecture. The performance overheads are obtained using an out-of-order model of the target processor and fault coverage results are obtained using an atomic model of the target processor.

The details of the processor configuration for out-of-order model used for the experiments are in Table II. We injected a total of 13000 faults per technique to evaluate the proposed solution, i.e. 1000 fault injection trials for each of the 13 benchmarks. Work by Leveugle et al. [26] can be used to calculate the statistical significance of the fault injection results. The calculation for our experimental setup shows that the with 95% confidence, margin of error for fault coverage results is 3.1%. After the fault injection, the program runs until completion. The result of each simulation trial is classified into one of the following five categories:

- **Masked**: The injected fault did not corrupt the program output. Application-level or architecture-level masking occurred in this case. Also faults that generate acceptable quality results are classified into this category.
- **HWDetect**: The injected fault produces a symptom such as a page fault so that a recovery can be triggered. A fault is considered under this category only if the symptom is produced within a number of cycles (1000 for our experiments) after the fault was injected.
- **SWDetect**: The injected fault was detected by the software checks inserted at the time of source code transformation.
- **Failure**: The injected fault resulted in out-of-bound address access and resulted in program termination. Also, faults causing infinite loops in the program are classified under this category.
- **USDC**: Faults that generate unacceptable data corruptions are classified into this category. These are the SDCs that do not have acceptable output.

D. Recovery Support

The proposed solution is a soft error detection-only solution. Once a soft error is detected, we rely on a recovery mechanism to recover from the detected error. Previously proposed solutions such as Encore [27], a software-only recovery scheme can be used for recovery. Checkpointing-based recovery schemes can also be used in conjunction with our solution. Moreover, previous works [6], [14] propose that in future processors, recovering from a checkpointed state of $\sim$1000 instructions would be required for aggressive performance speculation. Such a recovery scheme, if available, can also be integrated with our solution.

V. Experimental Evaluation and Analysis

Two of the most important parameters of any reliability scheme are its performance overhead and provided fault coverage. We obtained performance overhead and fault coverage results using the experimental setup described in the above section. We use the simulated runtime of the application as a performance measure and use this runtime to compare the performance of different techniques. Our overall technique is a combination of critical variable checks by duplication and value checks. To analyze the contribution of each of these, we present results for both.

Performance Overhead: Figure 12 shows the performance overhead measured in terms of runtime. *Dup only* column shows the performance overhead if the duplication of state variables is performed and no expected value checks are inserted. The mean performance overhead of *Dup only* is only 7.6%. *Dup + val chks* column for each benchmark shows the performance overhead if the duplication of the

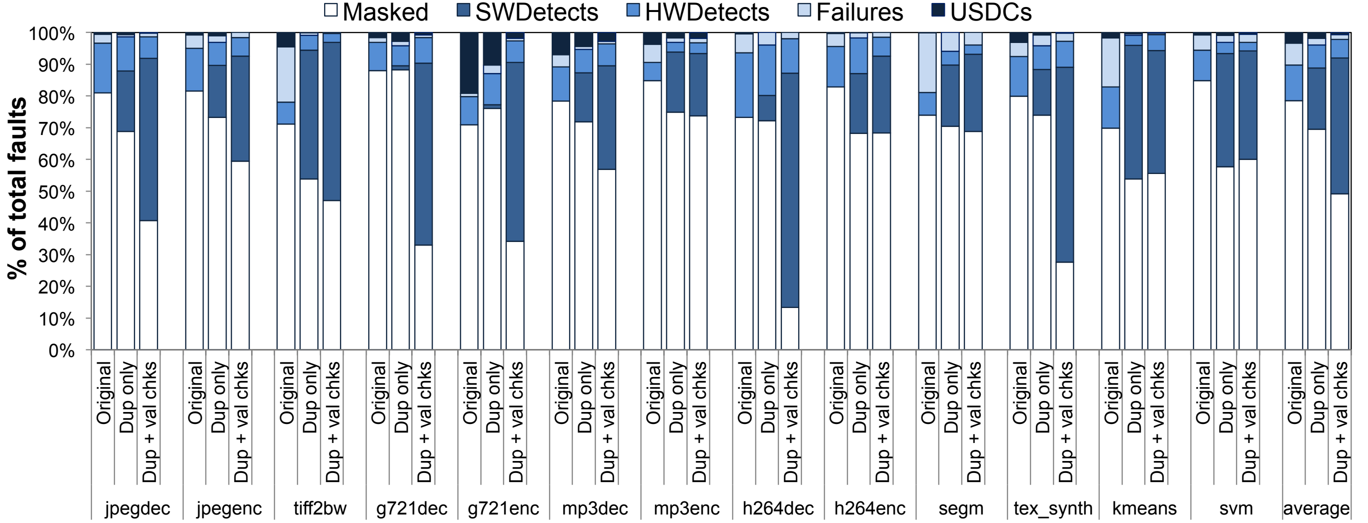

Figure 11: The fault outcome distribution among different categories is shown. Column *original* shows the distribution for original unmodified code. The fault distribution with code duplication and code duplication along with value checks is shown in *Dup only* and *Dup + val chks*, respectively.

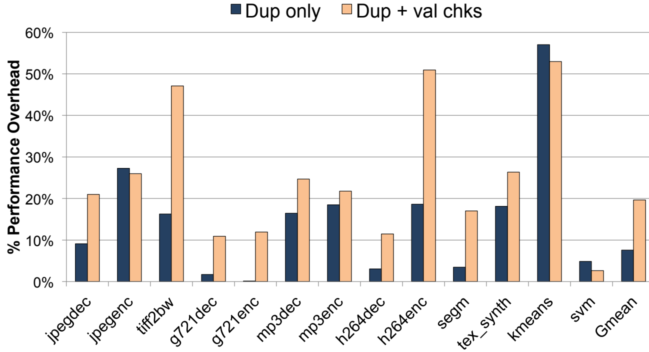

Figure 12: Performance overhead of checking by 1) duplicating the producer chain of state variables. 2) duplicating the producer chain as well as inserting value checks wherever necessary.

producer chains of state variables and expected value checks are inserted. *Dup + val chks* also includes the two optimizations described in Section III-C that arise out of the interaction between duplication and inserting value checks. The mean performance overhead for *Dup + val chks* is 19.5%. Four benchmarks *jpegdec, tiff2bw, mp3dec, h264dec and tex_synth* see a relatively bigger increase in performance overhead from *Dup only* because these benchmarks have a number of instructions amenable for value checks. It is interesting to note that the overhead of *svm* is lower for *Dup + val chks* than *Dup only* even though we found that the number of dynamic instructions are higher in *Dup + val chks*. This is due to the lower data cache misses and branch mispredicts in the later case. The average overhead of full duplication technique (not shown in Figure 12) also used by Khudia et al. [9] is 57% for the benchmarks used in our work. Full duplication is maximum amount of duplication possible without duplicating loads/stores.

Fault Coverage: We analyze the fault coverage results for unmodified, *Dup only* and *Dup + val chks* by injecting faults using the setup described in Section IV. If a fault results in *Masking*, *SWDetect* or *HWDetects*, the system can correctly execute the program. Hence, fault coverage is defined as the percentage of injected faults that result in Masking, SWDetect or HWDetects. First, faults are injected into original unmodified applications and their outputs are classified among *Masked, SWDetects, HWDetects, Failures and USDCs* based on the effect of the fault on the application execution. The results for this classification are shown in Figure 11. Y-axis in the figure plots the percent of total injected faults into an application. Results of fault injections into unmodified applications are shown in the first column (*Original*) for each benchmark. The *Original* column does not have any *SWDetects* because there are no software checks in the binary. Faults in unmodified applications generate 3.4% USDCs. Second, fault coverage of state variable only duplication is shown in *Dup only* column. It improves fault coverage for all the benchmarks and reduces SDCs and USDCs. *Dup only*, on average, has 1.8% USDCs. Finally, *Dup + val chks* column show the fault coverage if the duplication of state variables along with expected value checks and all optimizations are used. *Dup + val chks* has only 1.2% USDCs. We have also calculated the USDCs for full duplication and this is not shown in a already dense Figure 11. The USDCs rate for full duplication is 1.4% at 57% performance overhead. Please note that loads/stores are not duplicated in full duplication, hence there are a number of faults that escape detection. This result shows that selective duplication along with value checks is a more efficient way than soft computation unaware full duplication to protect soft computation workloads.

Acceptable SDCs: Another important analysis is the number of acceptable outputs among silent data corruptions. In this experiment, we break down SDCs further be-

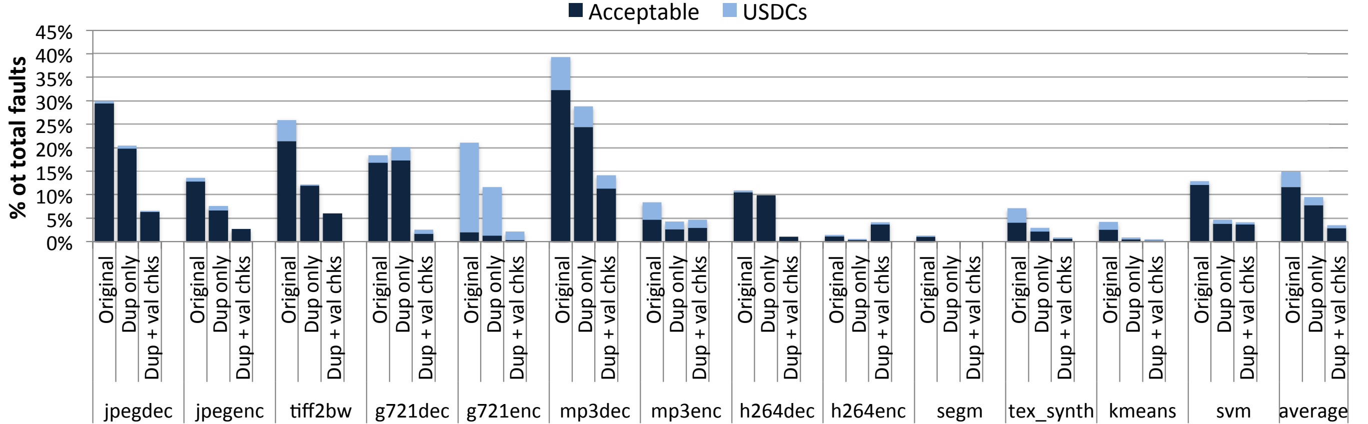

Figure 13: Each column represents the silent data corruptions as a percentage of total faults. The stacks in each column further divide the silent corruptions between acceptable program outputs and unacceptable data corruptions.

tween Acceptable SDCS (ASDCs) and Unacceptable SDCS (USDCs). Figure 13 shows the result of this analysis for unmodified, *Dup only* and *Dup + val chks*. Each column in Figure 13 represents the total number of SDCs for the corresponding benchmark. For example, for *kmeans* 4.2% of the total injected faults into the unmodified (*Original* column) application resulted in SDCs. Each column is further divided into ASDCS and USDCs. On average, SDCs are reduced from 15% down to 9.5% when going from *Original* to *Dup only* while USDCs are reduced from 3.4% down to 1.8% for the same and SDCs are reduced from 15% down to 7.3% when going from *Original* to *Dup + val chks* while USDCs are reduced from 3.4% down to 1.2% for the same. It is interesting to note that *mp3enc* and *h264enc* have higher SDCs for *Dup + val chks* than *Dup only*. This is due to the interaction of code duplication for state variables and expected value checks. An optimization (Optimization 2 in Section III-C) that we implemented to minimize performance overhead is to insert value checks instead of code duplication wherever beneficial in terms of performance overhead. This, however, in some cases can result in more SDCs if critical value checks are left out.

Sensitivity of results to different inputs: To ascertain the insensitivity of results to input variations, we performed 2-fold cross-validation on our results. We switched test and train inputs, i.e. test input was used to obtain profile data and train input was used in fault injection runs, to obtain fault coverage results for *Dup + val chks*. We performed cross-validation on *jpegdec* and *kmeans* from two separate fields. Cross-validation was performed only on these two benchmarks due to a large number of runs required for obtaining fault coverage results. The average performance overhead difference is 3%. The difference between *Masked*, *SWDetect*, *HWDetect*, *Failures* and *USDCs* is only .2%, .45%, .05%, .15% and .15%, respectively.

Impact of False Positives: A false positive occurs when one of the value checks fails at runtime in the absence of a fault. In such cases, an unnecessary recovery needs to be triggered. A high false positive rate increases the overhead due to unnecessary recoveries in a fault detection and recovery system. For pipeline-flush based recovery, Racunas et al. [13] calculate that 1 recovery initiation per 1000 instructions does not degrade the performance significantly (2% to 6%). This degradation in performance is dependent on the particular recovery technique. In comparison, in our case, the average false positive rate across all the evaluated benchmarks is 1 value check fail per 235K instructions. For our current implementation, the profiling is done only on one input but the false positive rate can be further reduced by combining profiling form multiple inputs and thus inserting checks only on more stable invariant values.

Comparison with prior work: Thomas et al. [21] define the notion of Egregious Data Errors (EDCs) for the outputs that deviate significantly from error-free outputs. Their work develops heuristics for placing detectors by analyzing the pointer and control data affected by a fault. In contrast, the main novelty of our scheme lies in the judicious combination of selective use of expensive duplication for critical state variables and inexpensive value checks for non-state parts of an application. The coverage of their scheme is measured assuming ideal (100% detection accuracy) detectors. Memory dependence (reaching stores for loads) in backward-slice-based detectors is not considered and hence coverage with actual detectors is expected to be lower. At 20% and 25% performance overhead (extra LLVM IR instructions) they report a 85% and 86% coverage of EDCs with *ideal* detectors, respectively. In comparison, even though it represents comparing IR instruction overhead and ideal detector coverage with runtime overhead and actual detector coverage, our technique shows 82.5% actual implemented detectors coverage of USDCs at 19.5% performance overhead (runtime).

VI. Related Work

Li et al. [8] propose the notion of application level correctness and also introduce the concept of acceptable

quality. We have used acceptable output quality measure in evaluating our work. The idea of user acceptable outputs has been used in previous research [5] to trade-off between energy efficiency/execution time and output solution quality. Li et al. [1] also previously proposed a light-weight recovery mechanism and soft instruction classification. They show a fault coverage of 96% with relaxed definition of correctness in comparison to 97.8% of fault coverage of our solution. Performance overhead of their technique is not reported in the paper. In comparison, we propose a technique to utilize the soft computing nature of applications and insert expected value checks to propose a low cost fault detection solution.

There are a few solutions proposed in the area of Software Implemented Hardware Fault Tolerance (SIHFT). SWIFT [28] uses instruction duplication in a single thread of execution. SWIFT protects the stores by duplicating their computation. However, the overhead of SWIFT is 1.41x even on an aggressive ILP-friendly Intel Itanium processor(more favorable for exploiting ILP offered by interleaved duplication). Our proposed solution only has 19.5% performance overhead on a ARM processor.

Shoestring [6] used the idea of protecting only global stores in order to lower performance overhead. Khudia et al. [9] improved Shoestring by utilizing profiling information. Both of these solutions, unlike our solution, do not use the notion of user acceptable outputs and do not incorporate application domain characteristics to increase the efficiency of their proposed solution. We compare with error detector placement work by Thomas et al. [21] in Section V. Sundaram et al. [29] propose selective replication of instructions that has 30% to 75% performance overhead. Cong et al. [30] propose an approach to protect instructions based on their criticality. The technique is a combination of static analysis and dynamic monitoring. The authors report energy savings and overhead of runtime monitoring but a combined performance overhead for duplication and runtime monitoring is not reported. Pattabiraman et al. [31] derived program level detectors using static analysis to find the best location for detectors to be placed in program to avoid system crashes. They identify certain properties such as fanout and lifetime from dynamic dependence graph of the program for detector placement. Unlike our work, their focus is not on reducing the large output corruptions but to avoid system crashes and in fault containment.

Likely program invariants have previously been used in checking validity of data streams, detecting software bugs [32], [33] and to lower SDC rates due to permanent hardware faults [34]. Range-checks used in this paper are also a form of likely invariants. However, we combine range-based checks with duplication to provide an effective transient fault detection solution. For transient faults (usually a single-bit upset), range-based checks should be frequent while permanent hardware faults continuously produce error hence sparing use of range-checks suffices. Our solution is optimized to have low-overhead even though relatively frequent checks are required to detect a single-bit upset. Other than high-cost high-reliability server class solutions such as DMR (Dual-Modular Redundancy) and TMR (Triple-Modular Redundancy), an approach to soft error reliability is Redundant Multithreading (RMT). Since processors which can execute multiple threads simultaneously are increasingly commonplace, the idea of using separate threads for error checking is a possibility. AR-SMT [35] introduced the idea of RMT on SMT cores. The actual work is done by a leading thread, and the trailing thread checks for the correctness. In comparison, our solution does not need to run any extra thread/process to provide fault detection.

There exist a number of hardware based solutions to provide protection against soft errors. In comparison, our solution is able to achieve high fault coverage with a low performance overhead without needing any specific hardware additions. Racunas et al. [13] present an hardware mechanism that can identify 85% of the injected faults to ensure that much of the program intermediate data falls within certain bounds. Their use of bounds on intermediate values in hardware is similar to our use of value checks in software. Argus [36] relies on a series of hardware checker units to perform online invariant checking to ensure correct application execution. Lee et al. [37] propose hardware-based scheme for partitioning failure critical and non-critical data into soft-error prone and soft-error protected caches. Soft error detection by anomalous microarchitectural behavior has been used by researchers to propose reliability solutions. Symptoms such as memory exceptions, branch mispredicts and cache misses are used in ReStore [14] to detect soft errors. These symptoms are an attractive way to detect soft errors at a relatively low cost. However, fault coverage starts to saturate as more symptoms are included and performance overhead starts rising. For example, using cache miss as a symptom might result in too many false detections. mSWAT [22] presented a solution that detects anomalous software behavior to provide a reliable system. mSWAT requires special simple hardware detectors to detect faults. Our solution, however, uses only the available symptoms such as page faults to classify faults under HWDetects category.

VII. Conclusions

The relentless pursuit of technology scaling in order to gain performance, energy efficiency and higher densities have made transistors more susceptible to soft errors. A growing number of applications from domains such as multimedia, computer vision, machine learning etc. do not need their output to be 100% correct. This numerically incorrect but acceptable output property of such applications can be exploited to provide an efficient fault tolerant solution.

In this paper, we propose a software-only solution that exploits the inherent fault tolerant nature of soft computing

applications. Our solution duplicates producer chains of certain critical variables and inserts expected value checks on other variables. We show that a combination of these two is helpful in reducing the number of unacceptable silent data corruptions. Overall, on average, SDCs are down from 15% to 7.3% and unacceptable SDCs are down from 3.4% to 1.2% in comparison to unmodified application. The performance overhead of the proposed technique is only 19.5% and it does not require any hardware modifications.

ACKNOWLEDGMENTS

This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We would like to thank the fellow members of the CCCP research group and the anonymous reviewers for their constructive comments and suggestions for improving this work.

REFERENCES

[1] X. Li and D. Yeung, “Exploiting soft computing for increased fault tolerance,” in *Workshop on Architectural Support for Gigascale Integration*, 2006.

[2] P. Shivakumar, M. Kistler, S. Keckler *et al.*, “Modeling the effect of technology trends on the soft error rate of combinational logic,” in *DSN*, Jun. 2002.

[3] B. T. Gold, J. C. Smolens, B. Falsafi, and J. C. Hoe, “The granularity of soft-error containment in shared memory multiprocessors,” *SELSE*, 2006.

[4] A. Sampson, W. Dietl, E. Fortuna *et al.*, “Enerj: Approximate data types for safe and general low-power computation,” in *ACM SIGPLAN Notices*. ACM, 2011.

[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” in *ASPLOS*. ACM, 2012.

[6] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, “Shoestring: Probabilistic soft-error reliability on the cheap,” in *ASPLOS*, Mar. 2010.

[7] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, “The soft error problem: An architectural perspective,” in *HPCA*. IEEE, 2005.

[8] X. Li and D. Yeung, “Application-level correctness and its impact on fault tolerance,” in *HPCA*. IEEE, 2007.

[9] D. S. Khudia, G. Wright, and S. Mahlke, “Efficient soft error protection for commodity embedded microprocessors using profile information,” in *LCTES*. ACM, 2012.

[10] S. K. S. Hari, S. V. Adve, and H. Naeimi, “Low-cost program-level detectors for reducing silent data corruptions,” in *DSN*. IEEE, 2012.

[11] M. Guthaus, J. Ringenberg, D. Ernst *et al.*, “MiBench: A free, commercially representative embedded benchmark suite,” in *Workshop on Workload Characterization*, Dec. 2001.

[12] B. Calder, P. Feller, and A. Eustace, “Value profiling,” in *MICRO*. IEEE, 1997.

[13] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee, “Perturbation-based fault screening,” in *HPCA*, Feb. 2007.

[14] N. J. Wang and S. J. Patel, “ReStore: Symptom-based soft error detection in microprocessors,” *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, Jun. 2006.

[15] Y. Ben-Haim and E. Tom-Tov, “A streaming parallel decision tree algorithm,” *The Journal of Machine Learning Research*, vol. 11, pp. 849–872, 2010.

[16] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems,” in *MICRO*, 1997.

[17] S. K. Venkata, I. Ahn, D. Jeon *et al.*, “Sd-vbs: The san diego vision benchmark suite,” in *IISWC 2009*. IEEE, 2009.

[18] J. E. Fritts, F. W. Steiling, and J. A. Tucek, “Mediabench ii video: Expediting the next generation of video systems research,” in *Electronic Imaging 2005*, 2005, pp. 79–93.

[19] T. Joachims, “Making large-scale svm learning practical,” 1999.

[20] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004.

[21] A. Thomas and K. Pattabiraman, “Error detector placement for soft computation,” in *DSN*. IEEE, 2013.

[22] S. Hari, M.-L. Li, P. Ramachandran *et al.*, “mswat: Low-cost hardware fault detection and diagnosis for multicore systems,” in *MICRO-42.*, dec. 2009.

[23] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, “Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline,” in *DSN*, Jun. 2004.

[24] D. S. Khudia and S. Mahlke, “Low cost control flow protection using abstract control signatures,” in *LCTES*. ACM, 2013.

[25] N. Binkert, B. Beckmann, G. Black *et al.*, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[26] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, “Statistical fault injection: quantified error and confidence,” in *DATE*. European Design and Automation Association, 2009.

[27] S. Feng, S. Gupta, A. Ansari *et al.*, “Encore: low-cost, fine-grained transient fault recovery,” in *MICRO*. ACM, 2011.

[28] G. Reis, J. Chang, N. Vachharajani *et al.*, “SWIFT: Software implemented fault tolerance,” in *CGO*, 2005.

[29] A. Sundaram, A. Aakel, D. Lockhart *et al.*, “Efficient fault tolerance in multi-media applications through selective instruction replication,” in *workshop on Radiation effects and fault tolerance in nanometer technologies*. ACM, 2008.

[30] J. Cong and K. Gururaj, “Assuring application-level correctness against soft errors,” in *ICCAD*. IEEE Press, 2010.

[31] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, “Application-based metrics for strategic placement of detectors,” in *Symposium on Dependable Computing*, Dec. 2005.

[32] M. D. Ernst, J. H. Perkins, P. J. Guo *et al.*, “The daikon system for dynamic detection of likely invariants,” *Proc. of the 2007 Science of Computer Programming*, 2007.

[33] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proceedings of the 24th international conference on Software engineering*. ACM, 2002.

[34] S. K. Sahoo, M.-L. Li, P. Ramchandran *et al.*, “Using likely program invariants for hardware reliability,” in *DSN*, 2008.

[35] E. Rotenberg, “AR-SMT: A microarchitectural approach to fault tolerance in microprocessors,” in *International Symposium on Fault Tolerant Computing*, 1999, pp. 84–91.

[36] A. Meixner, M. Bauer, and D. Sorin, “Argus: Low-cost, comprehensive error detection in simple cores,” *IEEE Micro*, 2008.

[37] K. Lee, A. Shrivastava, I. Issenin *et al.*, “Mitigating soft error failures for multimedia applications by selective data protection,” in *CASES*. ACM, 2006.