Low-cost Program-level Detectors for Reducing Silent Data Corruptions

Siva Kumar Sastry Hari,[†] Sarita V. Adve,[†] and Helia Naeimi[‡]
[†]Department of Computer Science, University of Illinois at Urbana-Champaign, swat@cs.illinois.edu
[‡]Intel Labs, Intel Corporation, helia.naeimi@intel.com

***Abstract*—With technology scaling, transient faults are becoming an increasing threat to hardware reliability. Commodity systems must be made resilient to these in-field faults through very low-cost resiliency solutions. Software-level symptom detection techniques have emerged as promising low-cost and effective solutions. While the current user-visible Silent Data Corruption (SDC) rates for these techniques is relatively low, eliminating or significantly lowering the SDC rate is crucial for these solutions to become practically successful.**

Identifying and understanding program sections that cause SDCs is crucial to reducing (or eliminating) SDCs in a cost effective manner. This paper provides a detailed analysis of code sections that produce over 90% of SDCs for six applications we studied. This analysis facilitated the development of program-level detectors that catch errors in quantities that are either accumulated or active for a long duration, amortizing the detection costs. These low-cost detectors significantly reduce the dependency on redundancy-based techniques and provide more practical and flexible choice points on the performance vs. reliability trade-off curve. For example, for an average of 90%, 99%, or 100% reduction of the baseline SDC rate, the average execution overheads of our approach versus redundancy alone are respectively 12% vs. 30%, 19% vs. 43%, and 27% vs. 51%.

***Keywords*-Hardware reliability; Transient faults; Silent data corruptions; Symptom-based fault detection; Application resiliency**

I. Introduction

Preserving hardware reliability is becoming increasingly challenging with technology scaling and increasing likelihood of in-field device failures even in commodity systems [1], [2]. Traditional circuit-level [3] and redundancy-based architecture-level [4], [5], [6] solutions have become too expensive in area, power, and performance for such systems, motivating very low-cost reliability solutions.

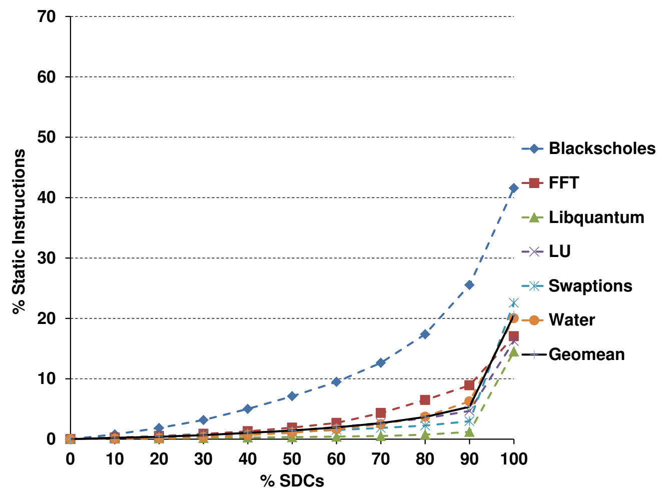
Symptom-based fault detection mechanisms [7], [8], [9], [10], [11], [12], [13] provide one such low-cost solution. These mechanisms treat anomalous software behavior as symptoms of hardware faults and detect them by placing very low-cost symptom monitors in hardware or software. Researchers have shown that this approach is effective in detecting both permanent and transient hardware faults [10], [11], with only a small fraction of faults escaping detection and producing silent data corruptions (SDCs). Faults resulting in SDCs produce corrupted application output without leaving any trace of failure behind, and represent the worst-case scenario for a resiliency solution. Although the SDC rate of symptom-based detectors is relatively small, it is still non-negligible and hard to bound, hindering the broad adoption of this approach in practice. This paper concerns systematically reducing the SDC rate due to transient faults in a cost-effective manner.

Whether a transient hardware fault will produce an SDC is highly dependent on the application; therefore, it is likely that the most cost-effective mechanism to reduce SDCs will be application-specific. The focus of this paper is on developing low-cost application-level checks (or detectors) that can detect potential SDC-causing faults. As the first step, we exploit recent work on identifying program sections that are susceptible to SDCs [14], [15]. Specifically, we use Relyzer [15], a technique capable of providing an application's complete instruction-level reliability profile. Relyzer selects a small fraction of application fault sites such that transient fault injections in these sites can estimate the outcomes of transient faults in all application sites. An application fault site refers to a combination of bit location and register operand of an executing instruction. We performed fault injections in the Relyzer-identified sites and obtained a comprehensive list of SDC-causing instructions in the entire application.

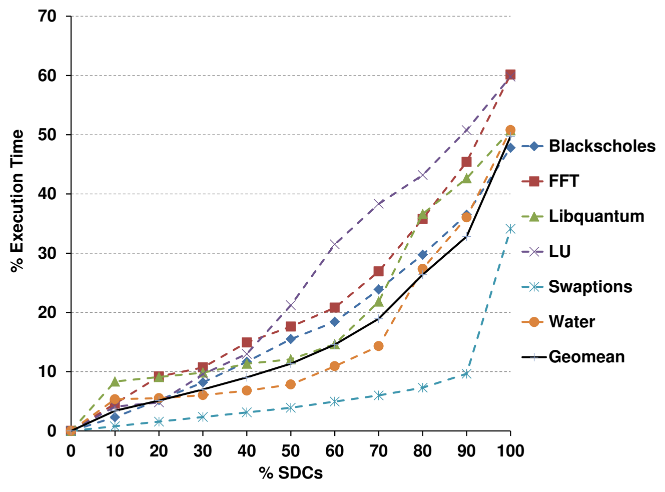
Investigating the SDC-causing fault sites revealed that only a small fraction of static instructions cause most SDCs. Figure 1(a) shows that virtually all the SDCs for the studied applications were caused by just 20.6% of the static instructions on average; 90% of the SDCs were caused by a mere 5.4% of the static instructions (Section III provides the detailed methodology for these results). This observation motivates using selective instruction-level detection techniques. Prior work has made similar observations, but has used selective instruction-level redundancy for detection [14], [16], [17].

Figure 1(b) shows the execution time (in number of dynamic instructions) consumed by the static instructions that cause SDCs. We find that the small fraction of SDC-causing static instructions consume a much higher fraction of the execution time. The figure shows that protecting all SDC-causing instructions through instruction-level redundancy may incur 50% overhead on average, assuming a conservative one cycle overhead per covered instruction (33% overhead on average for covering 90% of the SDCs).

This work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program) and the National Science Foundation under Grant CCF 0811693.



(a)



(b)

Figure 1. SDC-causing instructions and their impact on execution time. For a given application and input, part (a) shows the percentage of (executed) static instructions that cause a given percentage of silent data corruptions (SDCs). Part (b) shows the fraction of execution time (on a 1 IPC machine) taken by the static instructions in part (a) (for the given percentage of SDCs). For example, in FFT, 2% of the static instructions cause 60% of the SDCs and take 21% of the execution time. (The detailed methodology is in Section III.)]

This high overhead is consistent with that reported for previous selective instruction-based redundancy techniques, and motivates selective detection techniques that are much more cost effective than instruction-level redundancy.

Figure 1 clearly motivates the need for selective instruction-level fault detection mechanisms, but without (redundancy) checks on every SDC-causing static instruction. Thus, for effective low-cost detectors, we need to answer two key questions: *where* to place the detectors and *what* to detect. For placement, we use the insight that detectors should be placed in program locations where errors from many SDC-causing instructions propagate into a few quantities (or variables). For detectors in these locations, we exploit program-level properties that hold true for the few (potentially) error carrying quantities. In our work, we place our detectors at the end of loops and function calls that contain SDC-causing instructions, and detect errors by testing program-level properties on variables that are live at these points (e.g., comparing the outcomes of similar computations and known value equalities). Our results show that the program-level detectors we developed provide significantly better SDC coverage (reduction in SDCs) vs. performance tradeoffs than redundancy based detectors. Overall, the contributions of our work are as follows:

- **Understanding program properties that lead to SDCs:** We analyzed program instructions that cause >90% of the SDCs in each application. For these SDC hotspots, we identified a few program properties that appear repeatedly within the same application and even across different applications.
- **Developing low-cost program-level detectors:** Using the above analysis, we developed detectors that are placed at the end of loops and function calls. These detectors invoke program-level property checks on a few variables that potentially carry errors from a large number of SDC causing instructions. We find that our detectors provide an average SDC coverage of 84% with an average execution overhead of 10%.
- **Effective SDC coverage vs. execution overhead trade-off curves:** Using our low-cost detectors, we present continuous SDC coverage vs. execution overhead trade-off curves for our applications. These curves fall back to instruction-level redundancy for the sites that our program-level detectors cannot cover. Compared to similar curves with instruction-level redundancy alone, our approach yields much better execution overheads for all SDC coverage targets of interest on average; e.g., 12% vs. 30% overhead for 90% SDC coverage, 19% vs. 43% for 99% coverage, and 27% vs. 51% for 100% coverage. The ability to quantify such curves enables programmers and system designers to effectively tune for resiliency vs. overhead, allowing them to target any SDC coverage with the lowest cost combination of our detectors.

II. Analyzing SDC-Causing Program Sections and Developing Program-Level Detectors

With the goal of reducing and possibly eliminating the reliance on instruction-level redundancy, we focus on finding alternate low-cost program-level error detectors. Our approach is to move up from the instruction-level to understand the program behaviors and properties that are responsible for producing SDCs.

We start by identifying the SDC-causing fault sites by performing fault injection experiments in sites selected by

Relyzer. Details of this fault injection campaign are discussed in Section III. This campaign allows us to obtain a list of SDC-causing static instructions along with the number of potential SDCs each of these instructions can produce. We then sort these instructions in decreasing order of the number of SDCs they can produce, and analyze them in that order. For each instruction, we inspect the disassembled binary code around it to associate an application code (C code) section with it[1]. To our surprise, we observed a few code properties appearing repeatedly across different locations in the same application and even across different applications.

Given the SDC-causing sites, the next goal is to identify *where* to place the detectors and *what* detectors to use. For placement (where), the program locations should be selected such that many faults propagate to these points in a few variables. We used the end of loops and function calls that contain the SDC-causing instructions. For the detectors (what), we exploit a range of program-level properties: (1) comparing similar computations, (2) checking value equality, (3) range checks, and (4) performing mathematical tests. While devising these program-level detectors, we also ensure that they are low-cost.

Our approach of placing the detectors at the end of loops and function calls can potentially increase detection latencies because the errors are allowed to propagate until a detection point. However, these latencies can be tolerated by the state-of-the-art full-system checkpoint and rollback mechanisms [18], [19]. A further exploration of the relationship between such detection latencies and recovery is part of our future work.

The rest of this section describes the program code sections that we identified as SDC prone with examples, and explains the low-cost program-level detectors we devised to detects SDCs in these code sections using the above mentioned insights.

A. Incrementalization in loops

We observed that a significant fraction of SDC-causing fault sites directly affect computations in loops. These application sites often correspond to the loop index variables and/or addresses referring to array elements that are accessed in every loop iteration. For example, Figure 2(a) and (c) give the source and compiled code respectively for a single loop in the LU application from the SPLASH2 benchmark suite. Almost all instructions operating on integer registers in this code section were listed high in the sorted list of SDC-causing fault sites[2]. In particular, these faults alone produced over 50% of all the SDCs in LU. Faults in this compiled code can result in SDCs in the following two ways: (1) A fault affecting i can either terminate the loop early or cause it to go back in the iteration space. Since there is no loop-carried dependence, the latter effect will always result in masking the fault. (2) Faults in addresses A and B can result in detection if the faulty address is unallocated. If the faulty address points to a valid but incorrect memory location then the fault may be masked or result in an SDC. In this scenario, we observed that faults in several low-order bits in A and B resulted in SDCs because faulty addresses pointed to incorrect locations in arrays a and b.

Analyzing this code further, we observe that it uses the loop incrementalization optimization [20]. This optimization is typically applied on programs that perform computations on array elements in loops. Addresses to access these array elements must be computed in every iteration. This can be expensive if computed from scratch from the initial value (involving a multiplication and an addition). Modern compilers, therefore, apply the loop incrementalization optimization where the new value of the address is computed from the value in the previous iteration, involving just an addition. This optimization is shown to produce significant performance benefits for array-based codes [20]. Figure 2(b) and (c) show the assembly codes without and with the incrementalization optimization respectively for the C code shown in Figure 2(a).

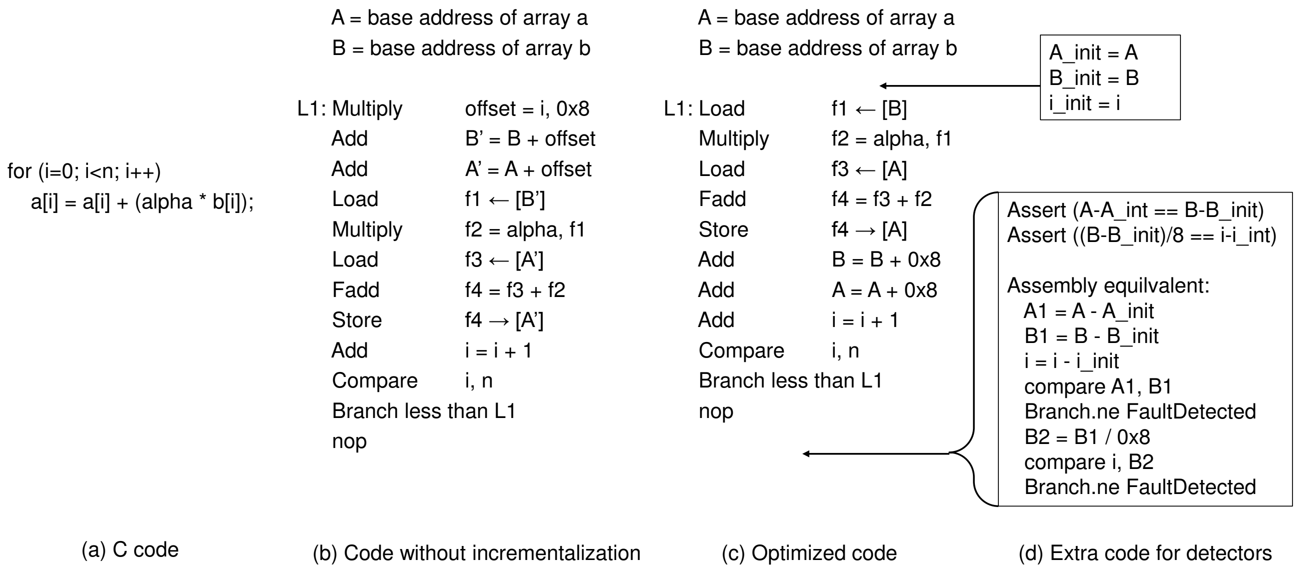
Detecting errors in incrementalized loops: Incrementalization makes errors in index variables and addresses used to access array elements propagate until the end of the loop. Hence, a property check at this location on these accumulated quantities can detect faults impacting these variables across all the iterations of this loop. Often the incrementalization in a loop is performed on multiple variables such that they all are incremented in every loop iteration with a value that is constant across iterations. We utilize this inherently similar computation to derive a property check at the end of the loop.

Figure 2(d) shows such a detector for our LU example. First, the initial values of A, B, and i are copied into different registers (or predefined memory locations). If the initial value of a register is predetermined as a constant then we can skip this step. For example, we do not have to collect the initial value of i because it is always 0. The values A and B are incremented with the same constant value in all the iterations. Hence the difference between their final and initial values should be the same. This property check can detect all single-event-upsets in these variables in all iterations of the loops. A similar check for variable i can also be performed by accounting for the different amount of increments used for i and A or B (also shown in Figure 2(d)). Since these detectors do not compromise coverage, we call them "lossless."

Codes that do not use the incrementalization optimization may produce intermediate values (offset, A', and B') in every loop iteration as shown in Figure 2(b). Since faults

[1]Compiler optimizations often make a direct association harder. However, we were able to identify the section of application code that contains the instruction of interest in most cases.

[2]Our fault model (explained in Section III) considers faults only in integer architecture register operands of executing dynamic instructions.



```
for (i=0; i<n; i++)
    a[i] = a[i] + (alpha * b[i]);
```

```
    A = base address of array a
    B = base address of array b

L1: Multiply        offset = i, 0x8
    Add             B' = B + offset
    Add             A' = A + offset
    Load            f1 ← [B']
    Multiply        f2 = alpha, f1
    Load            f3 ← [A']
    Fadd            f4 = f3 + f2
    Store           f4 → [A']
    Add             i = i + 1
    Compare         i, n
    Branch less than L1
    nop
```

```
    A = base address of array a
    B = base address of array b

L1: Load            f1 ← [B]
    Multiply        f2 = alpha, f1
    Load            f3 ← [A]
    Fadd            f4 = f3 + f2
    Store           f4 → [A]
    Add             B = B + 0x8
    Add             A = A + 0x8
    Add             i = i + 1
    Compare         i, n
    Branch less than L1
    nop
```

```
A_init = A
B_init = B
i_init = i
```

```
Assert (A-A_int == B-B_init)
Assert ((B-B_init)/8 == i-i_int)

Assembly equilvalent:
  A1 = A - A_init
  B1 = B - B_init
  i = i - i_init
  compare A1, B1
  Branch.ne FaultDetected
  B2 = B1 / 0x8
  compare i, B2
  Branch.ne FaultDetected
```

(a) C code
(b) Code without incrementalization
(c) Optimized code
(d) Extra code for detectors

Figure 2. An "SDC-hot" code section with loop incrementalization in LU from the SPLASH2 benchmark suite: (a) C code, (b) unoptimized assembly without loop incrementalization, (c) optimized assembly with loop incrementalization, and (d) detector for the optimized code. Faults in this (optimized) loop alone produce $>50\%$ of all SDCs in LU. The extra code in part (d) detects errors affecting i, A, and B in the optimized code. Initial values of these registers are collected at the beginning of the loop. These values are later used at the end of the loop to test the program-level properties.

affecting these intermediate values do not propagate to the end of the loop in a few variables, deriving a low-cost error detector is hard for non-incrementalized versions.

B. *Registers with long life*

We observed that a sizable chunk of SDCs were caused by faults in registers with long life, with multiple uses through this life. For example, we observed that the register holding the value n in Figure 2(c) is SDC prone. This register stays alive until the end of the loop and is used in every iteration of the loop. Other prominent examples are the registers that hold stack and frame pointers. These registers are typically set at the beginning of a function call and stay alive until the last instruction in the function body is executed.

Detecting errors in a register with long life: Errors in such a register remain alive until the end of the life of the register. Hence, the location to place a detector is, trivially, just after the last use of this register. If the register is used in many instructions through its life, then the cost of the detector is amortized across all of those uses. For this detector, we first attempt to identify another register or a constant such that its value can be compared to our target register. If this attempt fails, then we record the register's initial value (created at the definition of this register) in a different register (or a predefined memory location). At the detection location, we compare the initial value with the latest value in the register. An example of this is detecting faults in the register that stores the value of n in Figure 2(c). The value of n at the end of the loop can be tested with its earlier recorded value (from the beginning of the loop or its definition point). These detectors, like the previous ones, are also "lossless."

C. *Application-specific behavior*

For some applications, a large chunk of the SDC-causing fault sites belong to a few procedures. These procedures often do not have any side effects; i.e., the only output of the procedure is the return value. The exponential function from the math library, the BitReverse function from the FFT application from the SPLASH2 benchmark suite, and the RanUnif function (uniform random number generator) from the Swaptions application from the Parsec benchmark suite are few examples.

Detecting errors in the exponential function: A significant fraction of SDC-causing sites in Blackscholes and Water from Parsec and SPLASH2 benchmark suites respectively belong to the exponential function. The output of this function depends only on the input and no other previously stored data. All the errors created by the static fault sites in this function body, therefore, propagate through the output at the end of this function. We therefore place our detector at the end of the function.

Naively testing the output for correctness at this location can be expensive due to the nature of the function. We utilized a basic mathematical property of this function such that the errors can propagate through accumulating quantities over different invocations. This allows us to perform the test infrequently and still cover all the fault sites in these invocations.

From the definition of the exponential function, we know that $\exp(i1 + i2) = \exp(i1) \times \exp(i2)$ and $\exp(i1 - i2) = \exp(i1) \div \exp(i2)$, where $i1$ and $i2$ are inputs and $\exp(i1)$ and $\exp(i2)$ are outputs of two invocations respectively.

This property allows us to accumulate inputs using addition or subtraction and outputs using multiplication or division respectively. To detect errors, we re-execute this function with the accumulated input and compare its result with the accumulated output. The cost of this re-execution will be amortized across several invocations of this function. To detect errors in tolerable latencies, the frequency of the invocation of this detector can be dictated by the recovery solution (by specifying the tolerable detection latency).

Since a floating point operation on all hardware inherently generates an error and the exponential function on large or small inputs can exacerbate this error, we decided to apply this test only on relatively smaller inputs; i.e., when the absolute value of the input is < 25. For the remaining inputs, we rely on redundancy. We observed that very few invocations in our applications use inputs that are $\leq$ -25 and $\geq$ 25. Moreover, we use a combination of addition and subtraction on input such that the absolute value of the accumulated input is closer to zero and accordingly we use multiplication or division to accumulate output. This detector may show a loss in detection coverage if the error caused by the fault is within the estimated precision error of the floating point operations. We therefore call this detector "lossy."

Detecting errors in BitReverse function: In the FFT application from the SPLASH2 benchmark suite, nearly half of the SDC-causing sites belong to a function called BitReverse. This function takes an integer value as input and reverses its bits in the boolean representation. For example, if the input is 3 (0011), a 4-bit value, then the output should be 12 (1100).

The output of this function depends only on the input and no other previously stored data. Hence all the errors generated within this function body propagate through the output at the end of this function making it an ideal location for detector placement. Since this procedure does not show any accumulating behavior, we resort to checking parity on both the input and output. Since they both have the same number of bits set, the computed parities should match and detect faults that makes output and input differ by an odd number of bits. Naive software implementation for parity generation, however, can be expensive. One of the most optimized ways is to compute it in parallel [21] as shown in Figure 3(a). Another way is to use the parity flag in Intel 64 architectures that is generated on every logical and arithmetic operation on the low-order byte of the result. Figure 3(b) shows how this flag can be used to compute the parity of a 32-bit value. It uses the conditional move if parity is odd instruction, CMOVPO [22]. This detector may lose coverage if the corrupted output has a multi-bit error, and is therefore "lossy."

```
Input: V (32-bit value)          Input: V (32-bit value)
Output: P                        Output: P

V1     = V >> 16                               P   = 0
V      = V1 XOR V                              V   = V OR 0
V1     = V >> 8                  CMOVPO        P   = 1
V      = V1 XOR V                              V   = V >> 8
V1     = V >> 4                  CMOVPO        P   = 0
V      = V1 XOR V                              V   = V >> 8
V      = V AND 0xF               CMOVPO        P   = 1
P      = 0x6996 >> V                           V   = V >> 8
                                 CMOVPO        P   = 0

            (a)                              (b)
```

Figure 3. Efficient computation of parity on a 32-bit value. Part (a) uses parallel parity computation and part (b) uses the parity flag in Intel 64 architectures.

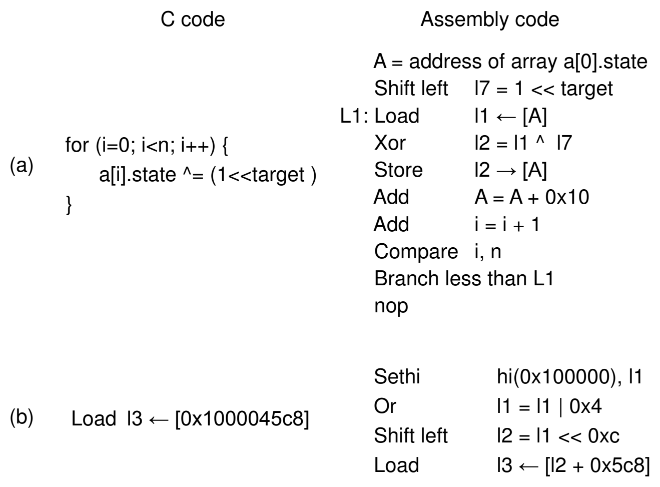
```
                C code               Extra code

KC=0;
for (K = 0; K < 9; K++) {
      Some computation
      if (condition)
            KC++;
}                    <----------  Assert (KC ≤ 9)
                                  Assert (K == 9)
```

Figure 4. A detector for a register with a fixed upper bound. The figure shows a code section from the Water application. Faults affecting this code eventually corrupt the value of KC and produce SDCs. The assertions show how these faults can be detected.

Detecting errors affecting registers with a fixed upper bound: A significant number of SDCs in the Water application from SPLASH2 were generated by errors in the variable KC in the code segment shown in Figure 4. To detect faults affecting the variables K and KC (directly and/or indirectly) in different iterations of this loop, we placed a detector at the end of the loop. From this code, it is evident that $KC \leq 9$ and $K = 9$ hold at the end of loop; we therefore used these invariants as detectors. Since all faults affecting K cannot be detected by testing $KC \leq 9$ alone, we also add $K = 9$ to the detector. Faults that affect KC alone (without corrupting K) such that $KC \leq 9$ may remain undetected. Since a loss in detection coverage can be observed, this detector is again "lossy."

Detecting errors in the random number generator from Swaptions: Over 90% of the SDCs in the Swaptions applications from the Parsec suite were caused just by a uniform random number generator function. This function takes a seed as the input and performs a series of integer operations to update the seed. This updated value is then used to generate the random number which ranges between 0 and 1. Since errors always propagate through the output, we place the detector at the end of this function call and it tests whether the output follows the specification; i.e., $0 \leq output \leq 1$. Since this detector cannot detect all the errors affecting the output of this function, it is "lossy."



```
          C code                                   Assembly code
                                           A = address of array a[0].state
                                           Shift left   l7 = 1 << target
                                       L1: Load         l1 ← [A]
     for (i=0; i<n; i++) {                 Xor          l2 = l1 ^ l7
(a)      a[i].state ^= (1<<target )        Store        l2 → [A]
     }                                     Add          A = A + 0x10
                                           Add          i = i + 1
                                           Compare  i, n
                                           Branch less than L1
                                           nop

                                           Sethi        hi(0x100000), l1
(b)   Load  l3 ← [0x1000045c8]             Or           l1 = l1 | 0x4
                                           Shift left   l2 = l1 << 0xc
                                           Load         l3 ← [l2 + 0x5c8]
```

Figure 5. SDCs due to local computations. (a) Code from the Libquantum application is shown where short-lived register values, $l1$ and $l2$, are created. Faults in these produce a non-negligible fraction of SDCs. (b) Instructions generated by the Sun cc compiler to compute a static address are shown. Again, faults in short-lived registers, $l1$ and $l2$, produce SDCs.

D. Local computations or registers with short life

We observed that a non-negligible fraction of SDCs were caused by faults in local computations with short register data flow chains. One example of this scenario is shown in Figure 5(a). Registers $l1$ and $l2$ store intermediate results and have short lives. Faults affecting these registers eventually corrupt the values stored in memory locations pointed by A. Another example of this pattern is the sequence of instructions that compute the static addresses known at compile time (Figure 5(b)). In SPARC V9 systems (our target machine), the global data section is stored above 1GB point in the virtual address space layout [23] and hence addresses of global variables require $>$32 bits. Multiple instructions are needed to generate these addresses because the ISA lacks instructions that can move constants of required sizes of $>$32 bits directly.

Since errors in the locally computed values do not propagate to a few values at an easily identifiable location in the program, deriving detectors and placing them for cost-effective detection is hard. Hence, we rely on instruction-level redundancy for these computations.

III. Experimental Methodology

We analyzed application resiliency by performing fault injection experiments in the fault sites that are selected by Relyzer [15]. Relyzer applies fault pruning techniques to select a small fraction of fault sites such that fault injection experiments in these few sites can estimate outcomes of all locations in the application.

For our fault model, we consider transient faults or single bit flips in various application fault sites. For each dynamic application instruction, every bit in each of its integer architecture register operands is considered as a separate fault site. Since this fault model considers fault sites that are highly likely to be architecturally live, it inherently filters a large fraction of masked faults (faults that do not affect application output). This allows us to focus more on faults that impact application output (and potentially cause SDCs). Our study does not consider faults in instructions from system calls and dynamically linked library function calls and faults in floating point registers; these are part of our future work.

We selected a mix of six applications from the SPLASH2 [24], Parsec [25], and SPEC CPU2006 [26] benchmark suites (Table I). All the selected applications were compiled using Sun C/C++ compiler version 5.9 with the highest level of optimization. We do not consider faults in the initialization and final phases of the applications where the inputs and outputs are read or written from/to files respectively and data structures are created and destroyed. We selected the inputs to the applications such that it was feasible to analyze 99% of all the fault sites reported by Relyzer. Overall we performed 890,000 fault injections across all the studied applications. These experiments were completed in approximately 3 days on a cluster of 175 compute nodes.

A. Fault injection framework

Our fault injection framework is based on Wind River Simics [27], an architecture level full system simulator. Our simulation environment models an UltraSPARC processor (using SPARC V9 ISA [28]) running a modern operating system (OpenSolaris). For fault injection, we first select Relyzer's specified injection locations which specify when and where to inject faults. In particular, an injection point is a tuple consisting a cycle number or instruction number (in our 1 IPC system), register number (that is either used as a destination or source in this instruction), and bit number (to flip the bit-value in that location). For each injection point, we start the application, execute it until the injection cycle number is reached, inject the specified fault, and run the application until a symptom detector (or program-level detector) is fired or the application output is produced. We use fatal traps, kernel panic, system error messages, checks for out-of-bounds memory accesses, and timeouts (executions taking more than twice the expected runtime) as symptom detectors [11], [15], [18]. As the last step, we compare the produced application output with the fault-free output to distinguish masked cases from SDCs.

B. Detectors and overhead evaluations

We implemented our program-level detectors (described in Section II) in Simics using breakpoints. Simics provides a framework to set breakpoints on various processor events and perform desired computations on these events. Our program-level detectors usually have two parts - one for collecting the information (typically at the beginning of

Application	Description	Input	Num. dynamic instructions after init & before finish phase	Num. executed static instructions after init & before finish phase
Blackscholes (PARSEC	Calculates prices of options with Black-Scholes partial differential equation	Sim-large	22.3 Million	538
FFT (SPLASH-2)	1D Fast Fourier Transform	64K points	548.0 Million	1,483
Libquantum (SPEC 2006)	Simulates a quantum computer running Shor's polynomial-time factorization algorithm	Test	235.4 Million	2,922
LU (SPLASH-2)	Factors a matrix into the product of a lower and upper triangular matrix	512 x 512 matrix 16 x 16 block	402.8 Million	1,124
Swaptions (PARSEC)	Computes prices of a portfolio of swaptions using Monte Carlo simulations	Sim-small	922.2 Million	1,696
Water-Spatial (SPLASH-2)	Evaluates forces and potentials that occur over time in a system of water molecules	512 molecules	504.4 Million	3,740

Table I
APPLICATIONS

	Operations	Estimated num instructions
Collecting a register value	$reg' = reg$	1
Lossless detectors	$r1 - r1' == r2 - r2'$	4
	$(r1 - r1')/const == r2 - r2'$	5
	$(r1 - r1')/r3 == r2 - r2'$	5
	$r1 == r2$	2
	$r1 == const$	2
	$r1 == r2 - const$	3
Lossy detectors	$r1 \leq const$	2
	Testing BitReverse functionality	20
	Accumulated check for exp function	20
	Range checking for RandUnif $0 \leq reg \leq 1$	4

Table II
EXTRA INSTRUCTIONS USED FOR MEASURING EXECUTION OVERHEAD

loops or functions) and the other for executing a specified check. At these points, we also collect information needed to measure the execution overheads. We measure the overheads in terms of the increase in the number of dynamic instructions. Table II shows the number of instructions we add to the application's total number of dynamic instructions on every invocation of collection or testing point of a detector. We measure the overheads for instruction-level redundancy by estimating that one instruction can be protected by one extra instruction even though the requirement is often more.

C. Evaluating the lossy detectors

The expected coverage of a detector is obtained by analyzing SDC causing sites and checking whether the detector can catch errors originating from these sites. Since the actual coverage observed by the lossy detectors may differ from the expected coverage, their effectiveness must be evaluated experimentally. Hence we performed a statistical fault injection campaign for the fault sites that are expected to be covered by these detectors. Overall we performed approximately 10,000 injections such that the error bars on our results are $< 2.8\%$ at 99% confidence level.

D. Determining the lowest overhead detectors for a target SDC coverage

Our detectors from Section II coupled with instruction-level redundancy based detectors provide a range of choices to achieve a given SDC coverage (fraction of SDCs detected). We would like to determine the lowest overhead set of detectors for each target SDC coverage, and understand the consequent trade-off between execution overhead and SDC coverage. Such SDC coverage vs. overhead curves also enable a fair comparison with instruction-level redundancy based detectors, allowing a comparison of performance overhead for a given target SDC.

To generate the above curves, we used a dynamic programming algorithm similar to one that solves the 0-1 knapsack problem. We start by labeling all (mutually exclusive) detectors of interest (redundancy based and/or our program-level detectors) with the fraction of SDCs they cover and their execution overheads (as discussed in Section III-B). We then run the optimization algorithm to find the combination of detectors with the minimum combined overhead with a constraint that the sum of the SDC coverage provided is at least equal to the target.

We generate execution overhead vs. SDC coverage curves for different classes of detectors: instruction-level redundancy only, our lossless detectors, and our lossless+lossy detectors. For the last two curves, some SDC causing static instructions of an application may not be covered (or may be only partially covered) by our program-level detectors. We therefore add instruction-level redundancy-based detectors for those static instructions to our dynamic programming problem. For the partially covered static instructions, the SDC coverage assigned to the redundancy based detectors (for the purposes of our optimization algorithm) is the number of SDCs not covered by our detectors. For the lossless+lossy curves, the dynamic programming algorithm assumes there is no coverage loss in the lossy detectors when

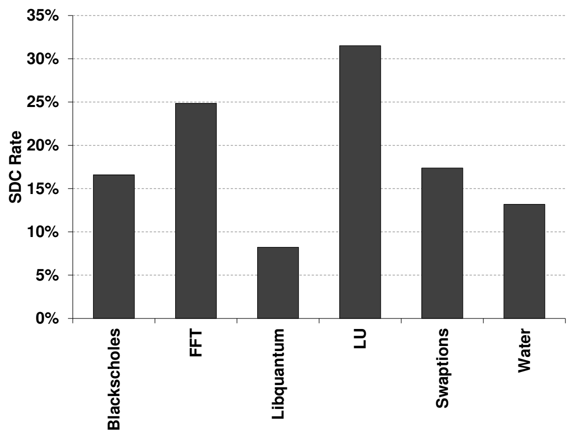


Figure 6. Baseline absolute SDC rates. These absolute rates are higher than previously reported for symptom-based detectors, largely because of the different fault models used.

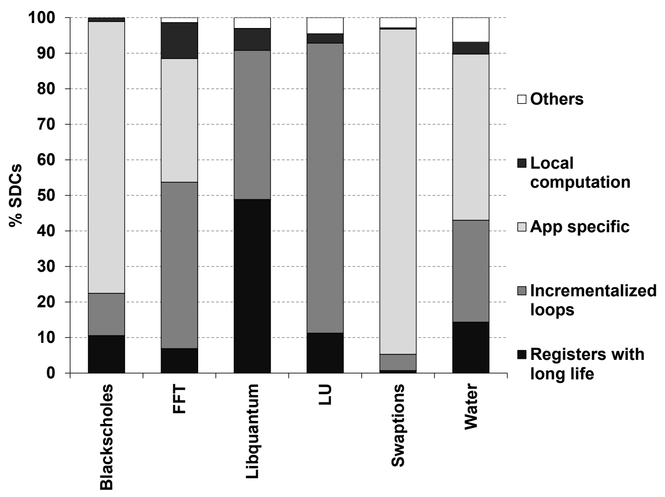


Figure 7. Contribution of code patterns from Section II to SDCs.

determining which redundancy based detectors to consider (but the SDC coverage attributed to the lossy detectors when plotting the curves does take into account the loss using the method in Section III-C). Thus, these curves may still terminate without covering all SDCs. Finally, the overall optimal solution for a target SDC coverage is to select the set of detectors that incur the least execution overhead among the above three trade-off curves.

IV. Results

A. Sources of SDCs

For reference, Figure 6 shows the absolute SDC rates obtained by our Relyzer-driven fault injection experiments as described in Section III. The SDC rates of our applications range between 8% to 32%. These are much higher than prior evaluations [11], [18], primarily because of the difference in the fault model. Our fault model considers faults in only those architectural registers that are highly likely to be alive, whereas prior work uses microarchitecture (and lower) level fault models which have a much higher masking rate [11], [29]. We chose the higher level fault model because our focus is on uncovering all possible SDCs with as few fault injections as possible and then reducing those SDCs. While we report the absolute SDC rate here for reference, the rest of this section focuses on the fraction of the baseline SDCs that are detected by our detectors (or SDC coverage). Comparing the absolute SDC rates for the different fault models is outside the scope of this paper.

To understand where in the program the SDC causing instructions come from, Figure 7 categorizes them based on the code patterns we identified in Section II. Figure 7 shows this categorization. We observe that fault sites that correspond to registers with long life and incrementalized loops produce a significant fraction of SDCs for FFT, Libquantum, LU, and Water (>90% of SDCs in Libquantum and LU). Application-specific behavior was a major contributor for Blacksholes, FFT, Swaptions, and Water. The figure shows that only a small fraction of SDC producing fault sites (up to 11.5%) were either categorized as local computations or not categorized at all (labeled as others in the figure) for all applications. This indicates that our detectors can potentially cover a large fraction of SDCs.

Applications	Num app. locations	Lossless		Lossy
		Loop based	Long lived register based	Application specific
Blackscholes	2	4	4	1
FFT	10	15	12	1
Libquantum	10	8	18	
LU	13	12	16	
Swaptions	9	12	5	1
Water	15	13	17	2

Table III
NUMBER OF DETECTORS PLACED IN THE STATIC APPLICATION CODE

B. Static overhead of the program-level detectors

Table III shows the program-level detectors placed in the static code for our applications. The second column shows the number of static application locations where our detectors were placed. The remaining columns show the number of detectors placed for covering faults in incrementalized loops, registers with long life, and application specific behavior. The sum of the last three columns may not add up to the value in the second column because multiple detectors can be placed in one static code location. The relatively small number of static code locations that require modifications shows that our devised detectors are not intrusive on the application. Moreover, the small number of application specific detectors means that limited program knowledge is required to implement them. This reinforces the benefit of Relyzer in pinpointing the SDC-vulnerable code sections that need examination.

C. *SDC coverage of the program-level detectors*

Since the program-level detectors were placed based on the SDC vulnerability of the fault sites, the corresponding reduction in the SDCs (SDC coverage) is known a priori, assuming that the added detectors are perfect. Thus, for the lossless detectors, the corresponding areas marked in Figure 7 (*incrementalized loops* and *registers with long life*) directly give the SDC coverage. We observe that on average, these detectors alone provide an SDC coverage of 50%. These detectors do not need further evaluation – they are sound and do not compromise coverage of their corresponding SDC sites.

Figure 7 shows that the application specific or lossy detectors also potentially cover a significant fraction of SDCs. Since these detectors can observe a coverage loss, their actual SDC coverage cannot be derived from their area in Figure 7. Instead we use a statistical fault injection campaign as explained in Section III-C. Our detectors for the exponential function, BitReverse function, values with upper bounds, and uniform random number generator show a coverage loss of 16%, 27%, 3%, and 33% respectively, relative to their expected or potential coverage indicated by Figure 7. For faults in the exponential function, we observed that most of the undetected faults produced outputs that could be tolerated by the application. For the random number generator, we observed that for our input set, the number of iterations of the corresponding Monte Carlo simulation executed is small and not yet convergent; preliminary experiments showed that with a large enough number of iterations, the errors may be tolerated in this case as well. In this work, however, we treat all undetected faults that result in output deviation as loss in coverage.

Figure 8 shows the total actual SDC coverage of our program-level detectors, combining both the lossy and lossless detectors. The figure shows that our detectors are highly successful, converting 67-92% of the original SDCs into detections (average of 84%), with both the lossy and lossless detectors contributing significantly.

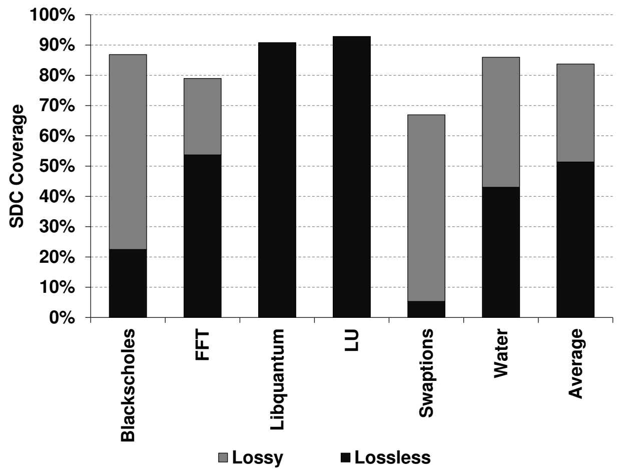


Figure 8. SDC coverage obtained by our program-level detectors, separated into coverage from the lossless and lossy detectors.

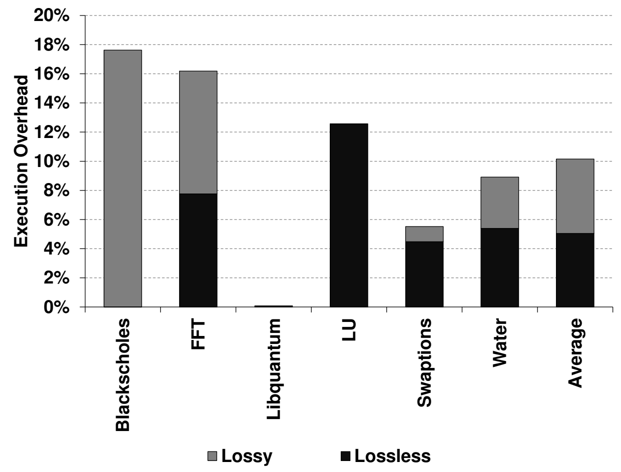


Figure 9. Execution overheads incurred by the program-level detectors, separated into coverage from the lossless and lossy detectors. The overhead of LU can be lowered to 3.4% with a small change in an input parameter without loss of performance or SDC coverage.

D. *Execution overhead from the program-level detectors*

Figure 9 shows the runtime overheads of our program-level detectors, separating the contributions from the lossy and lossless detectors. The overheads range from 0.08% to 18%, with an average of 10%.

The largest overheads come from the lossy application-specific detectors. Specifically, the exponential function in Blackscholes and the BitReverse function in FFT take the overheads for these applications to over 10%. Libquantum, Swaptions, and Water see much lower overheads of under 10%. Libquantum in particular sees almost zero overhead because of its use of loop-based detectors placed at the end of long running loops.

Although LU shows an overhead of 12.57%, a closer look showed that it can be lowered significantly. One of the loop based detectors (shown in Figure 2) executes with high frequency because the loop terminates after a small number of iterations (16 in particular). The number of iterations of this loop is dictated by a parameter that controls the block-size used by the blocking optimization for improving the effectiveness of memory hierarchies. This parameter can be increased to 64 on modern processors without any loss of performance [25]. When we deployed our detectors on this application with the block-size parameter set to 64, we observed that the overheads reduced to a much lower 3.24%. Since all the detectors used in this application are lossless, there is no compromise on SDC coverage with this modification.

E. *SDC coverage vs. execution overheads*

Figure 10 plots, for each application, SDC coverage vs. execution overhead trade-off curves for different classes of

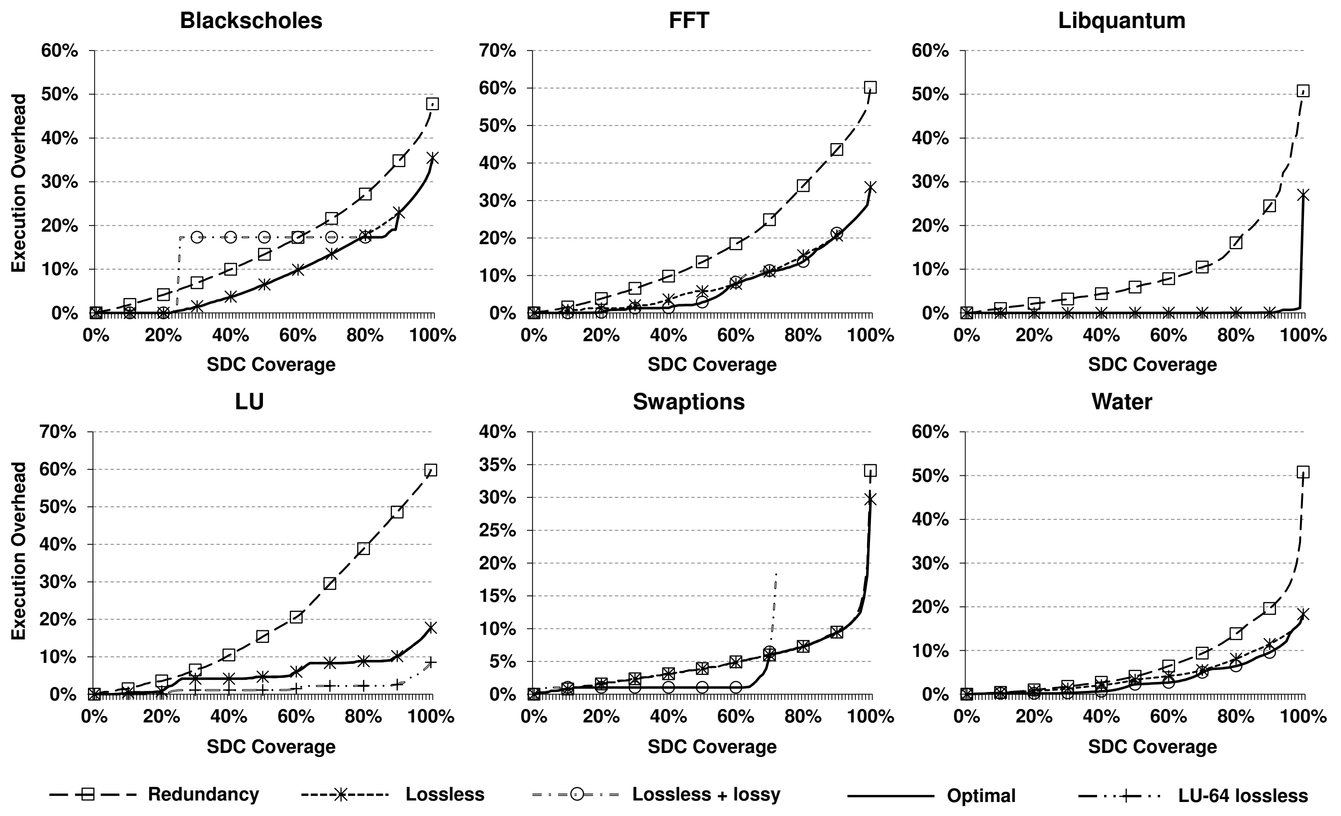


Figure 10. SDC coverage vs. execution overhead for each application for different classes of detectors.

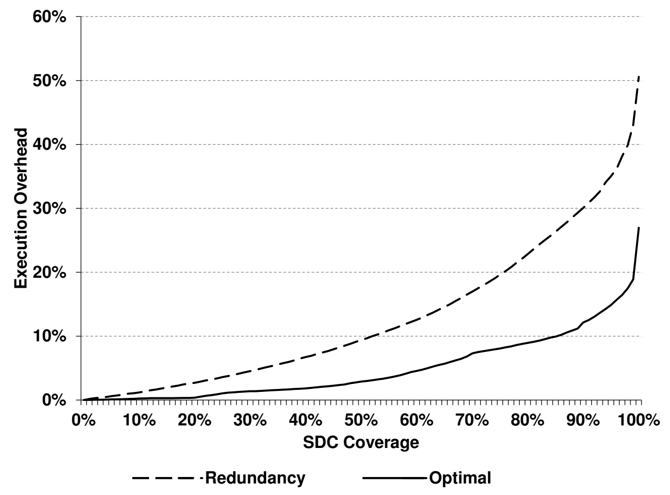


Figure 11. SDC coverage vs. execution overhead, averaged across all applications, for redundancy based and (optimal) program-level detectors.

detectors: instruction-based redundancy, lossless program-level, lossless+lossy program-level, and optimal that combines the best of the above. For LU, the figure also shows a curve for the version with the block size of 64 (using the same SDC profile as for the base LU since the application binary and inputs other than block size are unchanged). The methodology used is as described in Section III-D. In particular, the curves for program-level detectors add (selective) instruction-level redundancy for the SDC targets they cannot otherwise reach. Figure 11 summarizes the above information by averaging across all applications for the redundancy-based and optimal curves. The above curves serve two purposes: (1) they provide a fair way to compare the redundancy based and program-level detectors by allowing overhead comparisons for a fixed SDC coverage target and (2) they enable programmers and system designers to systematically trade off SDC coverage and performance.

The graphs show that our program-level detectors can reduce overhead relative to instruction-level redundancy alone at all target SDC coverage points for most of the applications. Focusing on Libquantum and LU, which do not use lossy detectors, we observe that in both cases, the overhead reduction relative to redundancy-only is quite high for the most part. The gains for LU are magnified when a larger block size of 64 is used (the "LU-64 lossless" curve). For Libquantum, the program-level detectors see near zero overheads to cover up to 91% of the SDCs. For both applications, the optimal curves fully overlap the program-level detector (lossless) curves.

Among the applications that use lossy detectors, all but Swaptions see significant overhead improvements for most of the interesting SDC coverage targets. In Blackscholes, the lossless+lossy curve shows a step behavior at 25% SDC coverage because the detector used to cover the SDCs in the exponential function with overhead of about 18% was required to achieve the target SDC coverage. This detector could have potentially capped the overhead for high SDC coverage points but its lossy behavior limited its coverage.

For FFT and Water, the use of the lossy detectors along with the lossless ones consistently provided lower execution overheads than lossless detectors alone. In Swaptions, the simple lossy detector provides a low-cost alternative to redundancy up to an SDC coverage of up to 70%. For higher coverage the optimal solution was, however, to use redundancy for the most part. The lossless detectors provided limited benefit in reducing the overhead needed to cover all SDCs.

Figure 11 shows that on average, our approach consistently yields much better execution overheads for all SDC coverage targets of interest. It shows that the optimal solution at 90%, 99%, and 100% average SDC coverage incur execution overhead of 12%, 19%, and 27% respectively, whereas the corresponding overheads for the redundancy-only solution are 30%, 43%, and 51% (which are 2.5X, 2.26X, and 1.89X higher).

V. Related Work

SWIFT [16] is a fully compiler-based software solution for fault detection. This technique inserts redundant code to compute duplicate versions of all register values, and validation code for checking the two versions. SWIFT more than doubles the number of dynamic instructions, relying on underutilized hardware resources for performance. CRAFT [17] later improved the performance of SWIFT through hardware support. PROFiT [30] improves upon both SWIFT and CRAFT by adding techniques to manage the desired levels of performance and reliability. It uses the program's performance and reliability profile (obtained by statistical fault injections) to identify the code sections that need duplication to meet the given performance and reliability constraints. Due to the lack of fine grained knowledge of the application's reliability profile, it considered duplication only at the function granularity. In our work, we obtain a detailed reliability profile through Relyzer and use selective redundancy only on the SDC causing instructions as our baseline. The focus of our work is to provide low-cost detectors such that higher reliability or performance can be achieved for a given performance or reliability budget respectively.

A more recently proposed technique called Shoestring [14] also shares our goal of reducing SDCs by protecting only those program instructions that potentially result in SDCs when subjected to transient faults. For identifying the potential SDC causing instructions, it employs a static program analysis that conservatively assumes that all writes to memory and function arguments are SDC causing sites. For protection, however, they rely on a SWIFT-like selective instruction-level redundancy based approach.

Symptom-based fault detection techniques [11], [10], [31], on the other hand, have emerged as low-cost alternatives for redundancy. These techniques have been shown to be effective in detecting a large chunk of transient and permanent faults with only a small fraction resulting in SDCs through statistical fault injections on microarchitecture-level models. Such techniques form the baseline for our work.

Range-based likely program invariants (inserted at stores) have been employed for detecting hard faults [12]. Results show a reduction in SDCs of up to 74% for a microarchitecture-level permanent fault model, but with an execution overhead of 14% on SPARC machines. Moreover, this technique suffers from false positives which can further increase the overheads. For a transient fault model, our technique provides a better SDC coverage vs. performance trade-off through a more selective placement of a broader range of detectors. Combining insights from these two studies for both fault models is part of our future work.

Pattabiraman et al. [32], developed metrics, namely *fanout* and *lifetime*, to identify what application variable to protect and where to place detectors. The goal, however, was to prevent or limit fault propagation and avoid system crashes with minimum possible detector locations, not particularly to reduce SDCs. Subsequently, they also proposed a technique to automatically derive application-specific detectors to be placed at these locations [9]. This technique tries to dynamically associate a property check for the identified variable from a set of pre-defined checks. The properties they used are similar in some respects to a few of our observations. However, they differ significantly because detectors in [9] never considered complex properties spanning across multiple variables like the loop based detectors presented in this paper. Moreover, detectors in [9] produce false positives, whereas our detectors never fire in fault-free executions.

VI. Conclusions and future work

With technology scaling, the hardware reliability problem is becoming increasingly challenging for a wide class of systems, motivating low-cost reliability solutions. Software-level symptom detection techniques have emerged as low-cost and effective solutions with low Silent Data Corruption (SDC) rates. However, eliminating or significantly lowering the user-visible SDC rate is crucial for these solutions to become practically successful. This paper presents an understanding of the program-level properties for a large fraction of SDC causing instructions. This analysis facilitated the development of low-cost program-level error detectors. We find that these detectors are able to convert an average of

84% of the SDCs to detections across our applications, at an average execution overhead of 10%. Compared to instruction-level redundancy alone, our program-level detectors (with instruction-level redundancy as backup) show, on average, significantly lower execution overheads at all SDC coverage targets of interest; e.g., 19% vs. 43% for 99% SDC coverage. Thus, the program-level detectors, owing to their lower cost and efficacy in detecting SDCs, provide practical and flexible choice points in the performance vs. reliability trade-off curve.

In the future, we plan to extend our study to more applications with expanded fault models and more accurate overhead estimates. Currently, the placement and derivation of the program-level detectors is manual. Ideally, we would like to automate this to the extent possible. In cases where application-specific knowledge is needed, we envision providing feedback to programmers such that they can make informed decisions to trade performance for reliability.

REFERENCES

[1] *International Technology Roadmap for Semiconductors*, http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf, 2009.

[2] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, 2005.

[3] Q. Zhou *et al.*, "Gate sizing to radiation harden combinational logic," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 155 – 166, Jan. 2006.

[4] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Intl. Symp. on Microarchitecture*, 1998.

[5] D. Bernick *et al.*, "NonStop Advanced Architecture," in *Intl. Conf. on Dependable Systems and Networks*, 2005.

[6] M. Mueller *et al.*, "RAS Strategy for IBM S/390 G5 and G6," *IBM Journal on Research and Development*, vol. 43, no. 5/6, Sept/Nov 1999.

[7] M. Dimitrov *et al.*, "Unified Architectural Support for Soft-Error Protection or Software Bug Detection," in *Intl. Conf. on Parallel Archtectures and Compilation Techniques*, 2007.

[8] O. Goloubeva *et al.*, "Soft-Error Detection Using Control Flow Assertions," in *International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[9] K. Pattabiraman *et al.*, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," in *European Dependable Computing Conference*, 2006.

[10] N. Wang *et al.*, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, July-Sept 2006.

[11] M.-L. Li *et al.*, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[12] S. Sahoo *et al.*, "Using Likely Program Invariants to Detect Hardware Errors," in *Intl. Conf. on Dependable Systems and Networks*, 2008.

[13] S. K. S. Hari *et al.*, "Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems," in *Intl. Symp. on Microarchitecture*, 2009.

[14] S. Feng *et al.*, "Shoestring: Probabilistic soft error reliability on the cheap," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[15] S. K. S. Hari *et al.*, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[16] G. Reis *et al.*, "SWIFT: Software Implemented Fault Tolerance," in *Proc. of Intl. Symp. on Code generation and optimization*. Washington, DC, USA: IEEE Comp. Society, 2005.

[17] G. Reis *et al.*, "Design and evaluation of hybrid fault-detection systems," in *Intl. Symp. on Computer Architecture*, 2005.

[18] P. Ramachandran, "Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment," Ph.D. dissertation, University of Illinois at Urbana Champaign, 2011.

[19] D. Sorin *et al.*, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," in *Intl. Symp. on Computer Architecture*, 2002.

[20] Y. Liu *et al.*, "Loop optimization for aggregate array computations," in *Proceedings of International Conference on Computer Languages*, May 1998, pp. 262 –271.

[21] "Bit Twiddle Hacks," Website, http://graphics.stanford.edu/~seander/bithacks.html.

[22] *Intel 64 and IA-32 Architectures Software Developer Manuals*, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[23] "Solaris 64-bit developer's guide," Website, http://docs.oracle.com/cd/E19253-01/816-5138/advanced-2/index.html.

[24] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Intl. Symp. on Computer Architecture*, 1995.

[25] C. Bienia *et al.*, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proc. of 5th Workshop on Modeling, Benchmarking and Simulation*, 2009.

[26] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, September 2006.

[27] Virtutech, "Simics Full System Simulator," Website, 2006, http://www.simics.net.

[28] D. L. Weaver and T. Germond, Eds., *The SPARC Arch. Manual*. Prentice Hall, 1994, version 9.

[29] M.-L. Li *et al.*, "Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults," in *Intl. Symp. on High Performance Computer Architecture*, 2009.

[30] G. Reis *et al.*, "Software-Controlled Fault Tolerance," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 4, 2005.

[31] P. Racunas *et al.*, "Perturbation-based Fault Screening," in *Intl. Symp. on High Performance Computer Architecture*, 2007.

[32] K. Pattabiraman *et al.*, "Application-based metrics for strategic placement of detectors," in *Proc. of 11th Pacific Rim Intl. Symp. on Dependable Computing (PRDC'05)*, 2005.