Article

An Ultra-Low-Cost Soft Error Protection Scheme Based on the Selection of Critical Variables

Yohan Ko

Division of Software, Yonsei University, Wonju 26493, Gangwon-do, Korea; yohan.ko@yonsei.ac.kr

Abstract: The exponentially increasing occurrence of soft errors makes the optimization of reliability, performance, hardware area, and power consumption one of the main concerns in modern embedded processors. Since the design cost of hardware techniques aimed at improving the reliability of microprocessors is quite expensive for resource-constrained embedded systems, software-level fault tolerance mechanisms have been proposed as an attractive solution for soft error threats. However, many software-level redundancy-based schemes are accompanied by considerable performance overhead, which is not acceptable for many embedded applications. In this work, we have introduced an ultra-low-cost soft error protection scheme for embedded applications, which works based on source-code analysis and identifying critical variables. After identification, these vital variables are adequately protected by placing runtime checks at critical points of execution. Our experimental results based on several applications demonstrate that the proposed scheme can mitigate the failure rate by 47% with negligible performance degradation.

Keywords: soft error; transient fault; fault tolerance; critical variable; embedded systems; protection technique

check for updates

Citation: Ko, Y. An Ultra-Low-Cost Soft Error Protection Scheme Based on the Selection of Critical Variables. *Electronics* **2021**, *10*, 2101. https://doi.org/10.3390/electronics10172101

Academic Editor: Tony Givargis

Received: 6 August 2021
Accepted: 28 August 2021
Published: 30 August 2021

1. Introduction

One of the primary sources of unreliability in modern processors is transient faults or soft errors. Soft errors can be caused by external sources, such as neutrons, alpha particles, and even cosmic rays, or internal events, such as power voltage noise Soft errors, can modify the state of a transistor or a memory element in the microprocessor. Although several masking effects at various levels have been introduced, ranging from the circuit level [1] to the software level [2] in contemporary microprocessors, it has been projected that the number of transient faults that can bypass the masking walls and change the final program output will increase by 30× as technology scales down from 45 to 16 nm technology [3,4].

In order to mitigate the problem of soft errors, researchers have proposed fault-tolerant mechanisms operating at several levels, including circuit level [5,6], microarchitectural level [2], and software level [7–14]. Among them, software-only approaches are attractive due to the ability to provide reliability without any hardware modifications [15] and the flexibility in application, i.e., high reliability only when required, such as in safety-critical systems.

Existing software-level soft error protection schemes can be classified into three broad categories: (a) redundancy based, (b) control flow checking, and (c) vulnerability reduction schemes. First off, the main idea behind the redundancy-based mechanisms [3,7–9,16–18] is to compute the program (or a part thereof) redundantly and then check for differences in the results. It has been shown that such a redundancy-based scheme can provide a high level of fault detection but, usually, there is significant performance overhead. For instance, the performance overhead of the state-of-the-art software-only full instruction duplication-based scheme is more than 200% [18].

In control flow checking mechanisms [10], predefined values (signatures) are written in a register at various points in the program. The register value is also checked at several points in the program to ensure that the execution flow is correct; else, an error is declared. Control flow checking approaches have moderate (around 20%–60%) performance overhead, but their fault coverage is very restricted and sometimes fails to protect [19]. The execution time of vulnerability reduction schemes changes the way a program executes on the processor so that it can exploit some inherent error masking effects. For example, in [12], the authors noted that the probability of fault masking in load and store instructions is higher than for other instructions. They labeled these instructions as critical and proposed to protect the processor using an instruction scheduling technique to prioritize the execution of critical instructions [12,13]. The current vulnerability reduction-based schemes are not selective in reducing the soft error exposure time of instructions and do not consider the criticality of the data. Therefore, they may reduce the soft error exposure time for non-critical data and increase the soft error vulnerability of the critical data.

In this work, we find the critical variable using a gemV framework [20], which is the vulnerability estimation toolset for microarchitectural components based on a cycle-accurate gem5 simulator [21]. Since gemV returns the reliability for microarchitectural components such as register file and memory, we have modified gemV to achieve the criticality of source-level variables. We then analyzed the condition of vulnerable variables, which are rarely updated but frequently read. For example, the variables that maintain the size of dimensions in an image processing application can be considered soft error critical. The reason is that if a soft error alters the value of such a variable, it will probably change the program output significantly.

We have exploited three protection schemes in order to protect such critical variables without hardware modifications. Firstly, the most obvious way to protect such variables is by using assertions at the high-level source code. However, one of the critical observations of this work is that many high-level source code assertions fail to detect soft errors, affecting processor registers which are holding critical variables. Since the memory subsystem, including the caches, is effectively protected by error correction codes (ECC) in many modern microprocessors, high-level assertions are not that effective. Secondly, we have used the "volatile" keyword for critical variable protection. If a variable is defined with the volatile keyword, the variable is excluded from the compiler optimization. Thus, the variable loads the value from the ECC-protected memory when needed for the processor execution.

We also introduce a novel software-only technique that inserts useful assertions at the assembly-level code. Notably, we find out which architectural registers hold the critical values at each point of time and add our low-level assertions right before the value of such registers become overwritten by other values or at the end of usage. Statistical fault injection experiments show that source-level assertions, volatile keywords, and assembly-level assertions can reduce the failure rate by 6%, 24%, and 47%, respectively, by just sacrificing less than 2% performance overhead.

The rest of this paper is structured as follows. In Section 2, we introduce related research regarding soft error protection techniques for embedded systems from the perspective of both hardware and software. Our solution, to choose the critical variable and apply protection schemes, is described in Section 3, and experimental setup and observations are summarized in Section 4. Finally, Section 5 concludes this paper and outlines the direction of future research.

2. Background and Motivation

With the advance of technology, embedded applications are becoming part of safety-critical and mission-critical applications. For instance, image processing has been used for obstacle detection and avoidance in autonomous vehicles. If an image processing algorithm produces the wrong results, it may cause a catastrophe such as a car accident. Soft errors or transient faults are one of the main reasons that can cause hardware malfunctions and ultimately jeopardize the functional safety of an application. Background radiation such as high-energy neutrons and protons are considered as the primary source of soft errors. Historically, soft errors were considered as a reliability challenge for high-altitude applications such as spacecraft and airplanes [22]. However, due to driving transistors being scaled down and near-threshold computing, soft errors threaten the reliability of even ground-level applications. The latest FinFET technology reduces the soft error rate to 7-nm scaling, but the continuous scaling to the 5-nm increases the per-bit soft error rate due to the critical charge [23]. Further, ITRS lists the soft error challenge as one of the most challenging reliability problems for ground-level applications in the near future [24].

Several schemes have been proposed to protect embedded architecture against soft errors, but they suffer from considerable overheads in terms of performance, area, and power consumption. The modular redundancy technique is one of the most straightforward techniques to protect systems against soft errors. The additional module performs precisely the same operations as the original module, and validation logic compares results from original and duplicated modules. If they are identical, there are no soft errors. If not, it can detect soft errors. If there is one more additional module, we can not only detect but also correct soft errors using the majority voting mechanism [25].

However, modular redundancy schemes need to modify hardware architecture, such as additional modules and validation logic, so it is inappropriate for resource-constrained embedded systems. Software-level redundancy schemes have been presented for soft error protection without hardware modification [26]. For instance, instructions are duplicated, and soft errors can be detected through additional "COMPARE" instructions before being used. Instruction duplication schemes can handle soft errors without expensive hardware area overheads but still suffer from the increasing number of instructions. Thus, they can cause massive overhead in terms of hardware area and performance even if all other variables are protected.

Fortunately, even if soft errors occur in microarchitectural components, they may not affect the final program output in many cases due to several masking effects [1,2] and OS-level soft error detection schemes [27]. For instance, a soft error can be masked if it is overwritten before being used. If a soft error incurs system halt, such as segmentation fault and page table fault, it can be detected by the operating system and an additional watchdog processor [28]. When a soft error causes an infinite loop, it can also be detected by cheap hardware protection.

However, it cannot be detected if a soft error results in incorrect output without OS-level error detection. We have analyzed the wrong outputs and realized that this is the issue in most cases, and the error alerts to specific values of a program, which we called critical values. Critical variables are the variables that are rarely overwritten but are frequently read. Therefore, by protecting critical variables in embedded applications, most of the undetectable failures can be covered without severe overheads.

Several types of research such as GVR [29], Rolex [30], RedThreads [31] proposed the high-level APIs to protect the most critical parts of applications such as variables or functions. However, they do not provide the way to find the most vulnerable parts of programs. They assume that software experts can have a high understanding of programs in order to define the criticality of variables and functions. However, it is challenging to define the vulnerabilities of each variable even for expert software engineers due to a large number of code lines nowadays. In this work, we have focused on the automated process to define the criticality of applications from both hardware and software perspectives.

3. Our Solution

We have analyzed high-level source code in order to effectively protect embedded applications without implementing hardware modifications. Our solution is to quantitatively find critical variables, and we effectively protect just the critical variables using software-based approaches. However, we encountered two main problems: (i) How can we choose critical variables against soft errors? (ii) How can we protect the chosen critical variables using software-only schemes?

3.1. How to Choose Critical Variables

In this paper, we have implemented a framework to analyze the criticality of source-level variables against soft errors as shown in Figure 1. First off, we use LLVM compiler [32] to create an executable from the high-level source code. We also modified the LLVM compiler to link the program-defined variables and physical registers. Then, we ran the executable from the LLVM compiler on the gemV simulator to achieve the vulnerability of physical registers. Lastly, our critical variable finder lists the vulnerability of all program-defined variables.

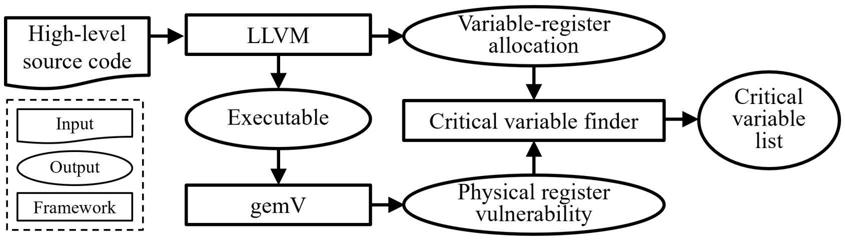


Figure 1. Overview of the high-level variable vulnerability estimation framework.

Initially, gemV can return the vulnerability of microarchitectural components against soft errors. Assume that a soft error flips the particular bit b in a microarchitectural component at the specific time t. When soft errors into (b, t) incur system failures, it is defined as vulnerable. If not (i.e., it returns the correct output without system malfunction), it is defined as non-vulnerable. The vulnerability is the sum of these vulnerable periods of microarchitectural components. Thus, the estimation can only show the hardware-level vulnerability, such as physical registers, and not software-level vulnerability, such as program-defined variables.

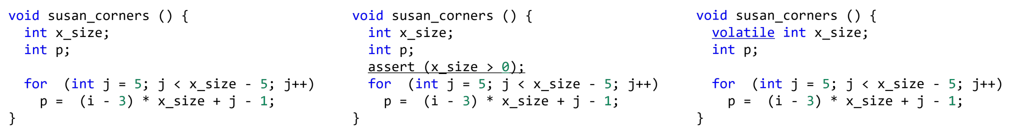
In order to link hardware-level registers and software-level variables, we have modified the LLVM compiler. We have assumed that all the source-level variables are stored in memory when they are defined. Moreover, the memory value is loaded when the variable is needed for computation in an application. Assume that we need to estimate the vulnerability of variable "A" in the function F1 against soft errors. First, the variable A is defined in function F1, and it is used for arithmetic computation. Then, the variable A is called by function F2, and it is used for the loop in function F1.

When variable A is defined in a high-level source code, register 1 (R1) is stored in memory address 0x0000. When used for the arithmetic computation, register 1 is vulnerable since it is read by ADD instruction in assembly-level analysis. The variable A is exposed based on high-level code analysis since the ADD instruction read the variable A. After the function F2 is called, the value in memory address 0x0000 (variable A) is loaded to R4. Since the number of registers is limited, R1 can be used in the function F2. Moreover, it means that R1 does not hold the variable A anymore after the function call. Since R4 is used for the loop in the function F1, R4 and variable are vulnerable.

Since the traditional definition of vulnerability is based on hardware-level components, gemV returns the vulnerability for register files such as R1 and R4. However, we have modified gemV in order to trace which register holds the specific variable at the particular time. Thus, we can achieve vulnerability for software-level variables, and we define the critical variables, which have more considerable vulnerabilities than other variables.

3.2. How to Protect Critical Variables

After identifying critical variables, the next question is how can we efficiently and effectively protect such variables against soft errors? For example, assume that we need to protect the variable "*x_size*" in a benchmark *susan*, especially from a function *susan_corners*, as shown in Figure 2. Figure 2a shows the original source code, which does not have protection schemes. The first and most common way to protect critical variables in a program is using high-level assertions. For example, if the value of some critical variables should be larger than zero during the execution time, one can add assertions for protecting such variables. Since the image size cannot be a negative value or zero, we can add an assertion that checks *x_size* is larger than zero Figure 2b.



```
void susan_corners () {
  int x_size;
  int p;

  for  (int j = 5; j < x_size - 5; j++)
    p =  (i - 3) * x_size + j - 1;
}
```

(**a**) Original high-level source code

```
void susan_corners () {
  int x_size;
  int p;
  assert (x_size > 0);
  for  (int j = 5; j < x_size - 5; j++)
    p =  (i - 3) * x_size + j - 1;
}
```

(**b**) High-level assertion to protect the variable *x_size*

```
void susan_corners () {
  volatile int x_size;
  int p;

  for  (int j = 5; j < x_size - 5; j++)
    p =  (i - 3) * x_size + j - 1;
}
```

(**c**) Volatile keyword to protect the variable *x_size*

Figure 2. Our high-level protection schemes to protect critical variables.

However, it is hard to insert sufficient and efficient assertions for critical variables. In the previous example, we have inserted an assertion to check whether *x_size* is larger than zero. Assume that the original value of *x_size* is 50. If it is changed to −50, it can be detected by the additional assertion. However, it cannot be detected if *x_size* value is changed to 150 due to soft errors. This is because the value is still larger than zero, even though it is corrupted. Further, we need to understand all the details of high-level source code for effective assertions.

Secondly, we can protect critical variables by defining them with the "volatile" keyword as shown in Figure 2c. The state-of-the-art compiler for embedded applications optimizes the high-level source code in order to reduce the energy consumption and performance overhead [33]. However, interestingly, compiler optimization can worsen the reliability against soft errors. Since data access to the register file is much faster than accessing cache or main memory, the compiler optimization tries to decrease the number of memory operations. We can use ALU instructions by allocating values in the variable in the register file with the optimization technique. However, it can easily be propagated to other components if there are soft errors in the frequently accessed registers.

On the other hand, the variables defined with a volatile keyword always access the memory when the values are required for computation. Since cache and main memory are protected by error correction code, even in tiny embedded processors, load instructions can load the correct values from the safe memory. Of course, the volatile keyword can induce severe performance overhead if programmers apply these techniques for too many variables. Based on our preliminary experiments, defining all variables with a volatile keyword can increase the runtime more than twice.

Lastly, we present low-level assertion by using register reservation and critical variable protection as shown in Figure 3. First off, we have compiled our protection target from high-level source code to low-level assembly code as described in Figure 3a. However, we cannot protect high-level variables from low-level assembly code since it has no link between physical-level register and software-level variables. So then, we have exploited the critical variable finder to link physical register index and software variables as depicted in Figure 3b. For instance, the first line "r3 (j)" means that the physical register r3 contains the variable *j*.

Then, we have applied low-level assertion from the assembly code by using register reservation as shown in Figure 3c. Since many embedded applications are not elaborate, they do not require many registers for execution. Moreover, several compilers, such as GCC and LLVM, support register reservation by using simple flags, i.e., *ffixed*. Our preliminary experiments show that the runtime is almost the same, even though we reserve one register out of 16 user-level registers.

We can then use these reserved registers to copy the value of critical variables without any hardware modifications. We have duplicated the value of critical variables by using data move ("MOV") instruction and reserved register r4 as depicted in the fourth line of Figure 3c. Note that the physical register r4 is used only to maintain the critical variable value in this protection. We have updated the value in the reserved register when the value of the critical value is changed. Then, we compare the original value of the data with that in duplicated registers before the memory is updated (“STORE”). Otherwise, we also compare the original value and duplicated one when they are used for the control flow change (“BNE”). If they are not identical, we can detect soft errors.

```
mov r3, #5
str r3, [fp, #-12]
ldr r3, [fp, #-16]

sub r3, r3, #5

ldr r2, [fp, #-12]


cmp r2, r3
bge .Loop_End
ldr r3, [fp, #-8]
sub r3, r3, #3
ldr r2, [fp, #-16]

mul r1, r3, r2
ldr r3, [fp, #-12]
add r3, r1, r3
sub r3, r3, #1
str r3, [fp, #-24]
                    ...

str r2, [fp, #-16]
```

(**a**) Original low-level assembly code

```
r3  (j)
r3  (j)
r3  (x_size)

r3  (x_size)

r2  (j)


r2  (j), r3 (x_size)

r3 (i)
r3 (i)
r2 (x_size)

r1 (p), r2 (x_size), r3 (i)
r3 (j)
r1 (p), r3 (j)
r3 (p)
r3 (p)


r2 (x_size)
```

(**b**) Pairing between physical registers and high-level variables

```
mov r3, #5
str r3, [fp, #-12]
ldr r3, [fp, #-16]
mov r4, r3
sub r3, r3, #5
sub r4, r4, #5
ldr r2, [fp, #-12]
cmp r3, r4
bne .Error_Detected
cmp r2, r3
bge .Loop_End
ldr r3, [fp, #-8]
sub r3, r3, #3
ldr r2, [fp, #-16]
mov r4, r3
mul r1, r3, r2
ldr r3, [fp, #-12]
add r3, r1, r3
sub r3, r3, #1
str r3, [fp, #-24]
                    ...
cmp r3, r4
bne .Error_Detected
str r2, [fp, #-16]
```

(**c**) Low-level assertion to protect the variable *x_size* based on the register reservation

Figure 3. Our low-level protection schemes to protect critical variables.

4. Experiments

4.1. Experimental Methodology

We have validated our methodology with three major experimental steps as shown in Figure 4. The first step is finding critical variables in high-level source codes using our critical variable finder based on the gemV framework [20]. We modified the gemV framework to trace architectural behaviors of register files and memory, and we can determine the vulnerabilities of variables in high-level C codes. Then, the critical variable finder transports the high-level source code and critical variable lists to the LLVM compiler.

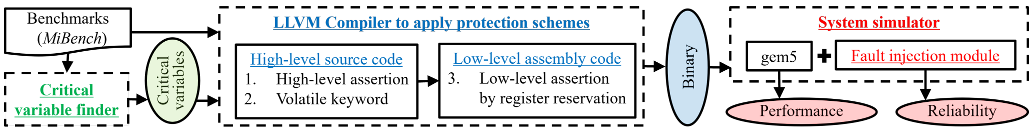


Figure 4. Our experimental setup to prove the efficacy of critical variable protection schemes.

The second step is applying protection schemes to the critical variable found from the first step. The high-level source code protection such as high-level assertion and volatile keyword can be easily applied to the high-level source code. We have modified the LLVM compiler to protect critical variables based on the low-level assertion. Our modified LLVM compiler reserves the additional register to protect critical variables and copies the value.

The last step is validating the performance and fault coverage of our software-based approaches, such as in terms of high-level assertion, volatile keywords, and low-level assertion. First off, we can estimate the performance based on the runtime from gem5 simulator. In this step, we also have performed statistical fault injection campaigns [34] for original applications and protected applications by high-level assertion, volatile keywords, and low-level assertion. We have injected 300 faults (i.e., soft errors) in the register file since errors on the register file can be propagated to other components.

After we have injected faults into microarchitectural components, we classify them as shown in Figure 5. If a program returns the correct output without any system malfunctions, it is defined as *masked*. If an application halts (e.g., segmentation fault, page table fault, system-defined exceptions, and etc.) due to the soft error, it is defined as *system halt*. If the runtime of an application exceeds double that of the original, we define it as *infinite loop*.

These two types of system failures can be easily captured by hardware-based protection schemes such as Watchdog timers [35].

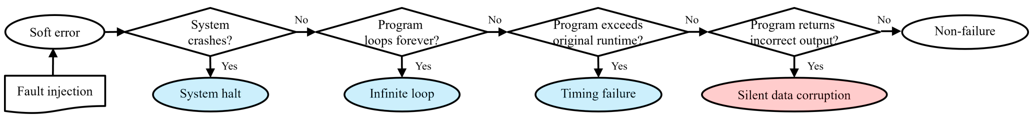


Figure 5. Our experimental setup to prove the efficacy of critical variable protection schemes.

Secondly, soft errors can result in the same output by varying the execution time slightly. If the deadline of applications is not essential, this type of system failure can be ignored. Our preliminary experimental results show that most timing failures vary the runtime within 10% compared to original execution. Further, even though applications need to meet the deadline, this type of system failure can be handled by Watchdog timers and scheduling approaches [36].

Lastly, soft errors can produce different program outputs without any symptoms of failures such as infinite loop and system halts. This type of system failure is called silent data corruption (SDC) since they are hardly found until programs finish. In the real world, silent data corruption can result in severe data loss and can require months of debugging time [37,38].

In order to estimate the fault coverage of fault detection mechanisms, we have only counted the number of SDCs. First off, we do not have to be concerned about masked faults since they do not affect the application. Furthermore, since the operating system can detect system halts and infinite loops, they are also excluded from the failure of error detection schemes. However, SDCs cannot be detected by OS or other hardware and software protections, so we need to decrease them using fault-tolerant techniques [39]. Further, we also consider the runtime overhead of our critical variable protection schemes since the it can increase the exposure to soft errors. Thus, we scale the number of silent data corruptions by multiplying it with the execution time which is normalized by the original application for fair comparison [40].

4.2. Experimental Results

What are critical variables? Our first set of experiments is the vulnerability estimation for variables of benchmarks in MiBench suite [41]. For a benchmark *susan (corners)*, the variable *max_no* in the function *susan_corners* is most vulnerable.

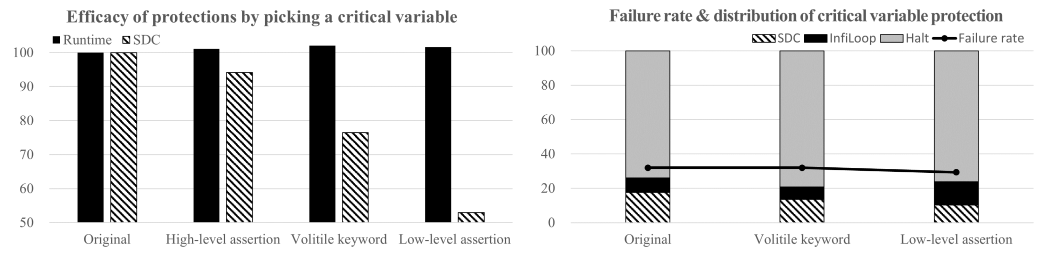
From the set of experiments, we have found the condition of critical variables. First off, the critical variable should be used in the majority of the runtime. For instance, the function *susan_corners* takes the runtime of the benchmark *susan (corners)*. Although a variable is critical in the function, which takes up 1% of the execution time, the maximum vulnerability is less than 1%.

Secondly, the critical variable should rarely be updated (or never updated). For instance, main function in the *susan (corners)* passes the the variable *max_no* to the function *susan_corners*. Then, the a value in *max_no* variable is fixed since it is used as the threshold in the loop. Even though there are soft errors in the register data, they are overwritten due to write operations.

Lastly, the critical variable should be frequently read. For instance, the variable *max_no* is used as the threshold value in the *susan_corners* function. Thus, the variable is read at every loop execution. If processors read the corrupted value in registers, the corrupted data can be propagated to the other components. Thus, the vulnerability should be large if there are many read accesses to the variable in the source code.

What is the benefit of protecting critical variables? Figure 6a shows the normalized runtime and number of SDCs by them of original applications (no protection) for a benchmark *susan* when we protect the most critical variable *max_no*. The high-level assertion, volatile keywords, and low-level assertion (register reservation) increase the runtime by about 2%. However, the reduction in failure rate depends on the protection techniques.

Using high-level assertion can decrease the SDC failures by only 6% as compared to the unprotected application. Through register reservation, volatile keywords and low-level assertion can decrease the SDC failures by 24% and 47%, respectively.



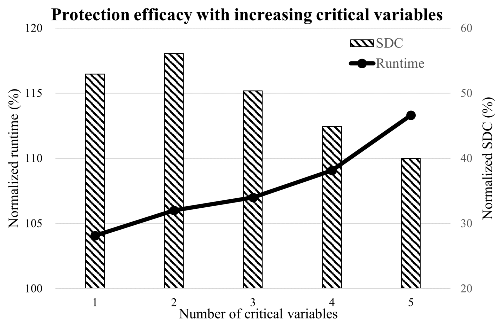
(**a**) Normalized runtime and number of SDCs to unprotected applications (**b**) Failure rate and its distribution with and without critical variable protection schemes

Figure 6. Our protection method, which picks a critical variable, is effective in terms of performance and SDC coverage.

Further, our method of critical variable protection is especially effective in mitigating SDC failures, which the operating system or hardware cannot catch. As we described in Section 4.1, the type of failures can be classified as silent data corruption (SDC), infinite loop (InfiLoop), and system halt (Halt). The failure rate from fault injection campaigns can be easily measured by dividing the number of total failures by the number of total injections. Figure 6b shows the failure rate and its distribution with and without applying critical variable protection schemes. Interestingly, the failure rate is similar regardless of applying protection techniques as shown in Figure 6b. However, the portion of each failure type varies upon the type of protection techniques as depicted in Figure 6b. For example, in the original application, SDC failure makes up 18% of total failures. On the other hand, only 14% and 10% of total failures are SDC failures with volatile keyword and low-level assertion protection schemes. This is because we chose the protection target from high-level source code rather than hardware-level components.

In order to show the availability of our critical variable protections, we have performed the fault injection campaigns with an increasing number of critical variables and various benchmarks as shown in Figure 7. First off, Figure 7a shows the normalized runtime and SDCs when we apply the low-level assertions for more critical variables. For example, if the system permits 15% performance overhead compared to the original application, we can reduce the 60% of SDCs by protecting five critical variables. Interestingly, the SDC coverage becomes worse when we protect two critical variables compared to the one variable protection. This is because two critical variable protection requires more execution time due to more reserved registers, but SDC coverage is similar to one variable protection. Since we have used the number of scaled SDC, which multiplies execution time and number of SDCs as described in Section 4.1, the fault coverage of two critical variable protection is worse than one critical variable.

Figure 7b shows the normalized runtime and SDC coverage as compared to original ones when we protect the most critical variable for various benchmarks. Note that e have chosen just one variable for each benchmark for simplicity, but we can increase the reliability by selecting more variables. On average, our low-level assertion reduces almost 30% of SDCs with just 1% runtime overhead. However, in the case of a benchmark *susan (corners)*, the runtime overhead is 4% since the most critical variable is chosen from the loop threshold, which contributes the majority of the runtime. Thus, the efficacy of our critical variable protection depends on the characteristics of applications, but it still mitigates at least 15% of SDCs with the negligible performance overhead for a benchmark *susan (smoothing)*.



(**a**) Normalized runtime and number of SDCs to unprotected applications when we increase the number of critical variables

Benchmarks	Runtime	SDC
ADPCM	100.12	58.53
FFT	100.74	74.84
IFFT	102.49	79.99
susan (corners)	104.06	55.09
susan (edges)	100.80	67.20
susan (smoothing)	100.12	85.50
Average	101.39	70.19

(**b**) Normalized runtime and number of SDCs to unprotected applications when we apply low-level assertions to various benchmarks

Figure 7. Our critical variable protection can have availability in terms of the number of critical variables and characteristics of applications.

5. Conclusions

Since soft errors are one of the most critical design concerns at the early design phase due to aggressive technology scaling and near-threshold computing, several protection schemes have been proposed to address them. However, soft error protection is not cheap in terms of performance, hardware area, and power consumption. In order to protect resource-constrained embedded systems, we proposed software-implemented schemes by choosing critical variables in the high-level source code. First, we estimated the criticality of software-level variables quantitatively by exploiting vulnerability estimation for microarchitectural components. Then, we protected just the critical variables using volatile keywords, assertion, and register reservation. Our experimental results show that we can decrease the failure rate by up to 47% with almost negligible overhead (<2%).

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Shivakumar, P.; Kistler, M.; Keckler, S.W.; Burger, D.; Alvisi, L. Modeling the effect of technology trends on the soft error rate of combinational logic. In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2002), Washington, DC, USA, 23–26 June 2002; pp. 389–398.
2. Mukherjee, S.S.; Weaver, C.; Emer, J.; Reinhardt, S.K.; Austin, T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In Proceedings of the IEEE 36th Annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, USA, 5 December 2003; p. 29.
3. Feng, S.; Gupta, S.; Ansari, A.; Mahlke, S. Shoestring: Probabilistic soft error reliability on the cheap. *ACM SIGARCH Comput. Archit. News* **2010**, *38*, 385–396. [CrossRef]
4. Dixit, A.; Wood, A. The impact of new technology on soft error rates. In Proceedings of the 2011 IEEE International Reliability Physics Symposium (IRPS), Monterey, CA, USA, 10–14 April 2011; p. 5B. 4.1–5B. 4.7.
5. Avirneni, N.D.P.; Somani, A.K. Low overhead soft error mitigation techniques for high-performance and aggressive designs. *Comput. IEEE Trans.* **2012**, *61*, 488–501. [CrossRef]
6. Nicolaidis, M.; Torki, K.; Natali, F.; Belhaddad, F.; Alexandrescu, D. Implementation and validation of a low-cost single-event latchup mitigation scheme. In Proceedings of the IEEE Workshop on Silicon Errors in Logic–System Effects (SELSE), Stanford, CA, USA, 27–28 March 2009.
7. Oh, N.; Mitra, S.; McCluskey, E.J. ED 4 I: Error detection by diverse data and duplicated instructions. *Comput. IEEE Trans.* **2002**, *51*, 180–199. [CrossRef]
8. Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I. SWIFT: Software implemented fault tolerance. In Proceedings of the IEEE International Symposium on Code Generation and Optimization, San Jose, CA, USA, 20–23 March 2005; pp. 243–254.
9. Zhang, Y.; Lee, J.W.; Johnson, N.P.; August, D.I. DAFT: Decoupled acyclic fault tolerance. *Int. J. Parallel Program.* **2012**, *40*, 118–140. [CrossRef]
10. Goloubeva, O.; Rebaudengo, M.; Reorda, M.S.; Violante, M. Soft-error detection using control flow assertions. In Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Boston, MA, USA, 5 November 2003; pp. 581–588.
11. Vemu, R.; Abraham, J. CEDA: Control-flow error detection using assertions. *IEEE Trans. Comput.* **2011**, *60*, 1233–1245. [CrossRef]

12. Rehman, S.; Shafique, M.; Henkel, J. Instruction scheduling for reliability-aware compilation. In Proceedings of the ACM 49th Annual Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012; pp. 1292–1300.
13. Rehman, S.; Shafique, M.; Kriebel, F.; Henkel, J. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Taipei, Taiwan, 9–14 October 2011; pp. 237–246.
14. Lee, J.; Shrivastava, A. Software-based register file vulnerability reduction for embedded processors. *ACM Trans. Embed. Comput. Syst. (TECS)* **2013**, *13*, 38. [CrossRef]
15. Reis, G.A.; Chang, J.; Vachharajani, N.; Mukherjee, S.S.; Rangan, R.; August, D. Design and evaluation of hybrid fault-detection systems. In Proceedings of the IEEE 32nd International Symposium on Computer Architecture (ISCA'05), Madison, WI, USA, 4–8 June 2005; pp. 148–159.
16. Reis, G.A.; Chang, J.; August, D.I. Automatic instruction-level software-only recovery. *IEEE Micro* **2007**, *27*, 36–47. [CrossRef]
17. Khudia, D.S.; Wright, G.; Mahlke, S. Efficient soft error protection for commodity embedded microprocessors using profile information. In Proceedings of the ACM SIGPLAN Notices, Philadelphia, PA, USA, 25–27 January 2012; Volume 47, pp. 99–108.
18. Didehban, M.; Shrivastava, A. nZDC: A compiler technique for near Zero Silent Data Corruption. In Proceedings of the ACM 53rd Annual Design Automation Conference, Austin, TX, USA, 5–9 June 2016; p. 48.
19. Shrivastava, A.; Rhisheekesan, A.; Jeyapaul, R.; Wu, C.J. Quantitative Analysis of Control Flow Checking Mechanisms for Soft Errors. In Proceedings of the ACM 51st Annual Design Automation Conference on Design Automation Conference, San Francisco, CA, USA, 1–5 June 2014; pp. 1–6.
20. Tanikella, K.; Koy, Y.; Jeyapaul, R.; Lee, K.; Shrivastava, A. gemV: A validated toolset for the early exploration of system reliability. In Proceedings of the 2016 IEEE 27th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), London, UK, 6–8 July 2016; pp. 159–163. [CrossRef]
21. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The gem5 simulator. *ACM SIGARCH Comput. Archit. News* **2011**, *39*, 1–7. [CrossRef]
22. Mukherjee, S.S.; Emer, J.; Reinhardt, S.K. The soft error problem: An architectural perspective. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA, 12–16 February 2005; pp. 243–247. [CrossRef]
23. Narasimham, B.; Chaudhary, V.; Smith, M.; Tsau, L.; Ball, D.; Bhuva, B. Scaling Trends in the Soft Error Rate of SRAMs from Planar to 5-nm FinFET. In Proceedings of the 2021 IEEE International Reliability Physics Symposium (IRPS), Monterey, CA, USA, 21–25 March 2021; pp. 1–5. [CrossRef]
24. IRC. International Technology Roadmap for Semiconductors 2.0-Executive Summary. 2015. Available online: http://www.itrs2.net/itrs-reports.html (accessed on 30 June 2015).
25. Arifeen, T.; Hassan, A.S.; Lee, J.A. A fault tolerant voter for approximate triple modular redundancy. *Electronics* **2019**, *8*, 332. [CrossRef]
26. Ko, Y.; Kim, S.; Kim, H.; Lee, K. Selective Code Duplication for Soft Error Protection on VLIW Architectures. *Electronics* **2021**, *10*, 1835. [CrossRef]
27. Li, M.L.; Ramachandran, P.; Sahoo, S.K.; Adve, S.V.; Adve, V.S.; Zhou, Y. Understanding the propagation of hard errors to software and implications for resilient system design. *ACM SIGARCH Comput. Archit. News* **2008**, *36*, 265–276. [CrossRef]
28. Ersoz, A.; Andrews, D.; McCluskey, E. The watchdog task: Concurrent error detection using assertions. *NASA Tech. Rep.* **1985**
29. Fujita, H.; Dun, N.; Fang, A.; Rubenstein, Z.A.; Zheng, Z.; Iskra, K.; Hammond, J.; Dubey, A.; Balaji, P.; Chien, A.A. Using global view resilience (GVR) to add resilience to exascale applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14), New Orleans, LA, USA, 16–21 November 2014.
30. Hukerikar, S.; Lucas, R.F. Rolex: Resilience-oriented language extensions for extreme-scale systems. *J. Supercomput.* **2016**, *72*, 4662–4695. [CrossRef]
31. Hukerikar, S.; Teranishi, K.; Diniz, P.C.; Lucas, R.F. Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *Int. J. Parallel Program.* **2018**, *46*, 225–251. [CrossRef]
32. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the IEEE International Symposium on Code Generation and Optimization (CGO 2004), San Jose, CA, USA, 20–24 March 2004; pp. 75–86.
33. Simunic, T.; Benini, L.; Micheli, G.D.; Hans, M. Source code optimization and profiling of energy consumption in embedded systems. In Proceedings of the 13th International Symposium on System Synthesis, Madrid, Spain, 20–22 September 2000; pp. 193–198. [CrossRef]
34. Ramachandran, P.; Kudva, P.; Kellington, J.; Schumann, J.; Sanda, P. Statistical Fault Injection. In Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN), Anchorage, AK, USA, 24–27 June 2008; pp. 122–127. [CrossRef]
35. Murphy, N.; Barr, M. Watchdog timers. *Embed. Syst. Program.* **2001**, *14*, 79–80.
36. Zhou, J.; Sun, J.; Zhang, M.; Ma, Y. Dependable Scheduling for Real-Time Workflows on Cyber–Physical Cloud Systems. *IEEE Trans. Ind. Inform.* **2021**, *17*, 7820–7829. [CrossRef]
37. Dixit, H.D.; Pendharkar, S.; Beadon, M.; Mason, C.; Chakravarthy, T.; Muthiah, B.; Sankar, S. Silent Data Corruptions at Scale. *arXiv* **2021**, arXiv:2102.11245.

38. Li, G.; Pattabiraman, K.; Hari, S.K.S.; Sullivan, M.; Tsai, T. Modeling Soft-Error Propagation in Programs. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; pp. 27–38. [CrossRef]
39. Yang, N.; Wang, Y. F_Radish: Enhancing Silent Data Corruption Detection for Aerospace-Based Computing. *Electronics* **2021**, *10*, 61. [CrossRef]
40. Schirmeier, H.; Borchert, C.; Spinczyk, O. Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors. In Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro, Brazil, 22–25 June 2015; pp. 319–330. [CrossRef]
41. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the 2001 IEEE International Workshop on Workload Characterization (WWC-4), Austin, TX, USA, 2 December 2001; pp. 3–14.