J Electron Test (2014) 30:193–212
DOI 10.1007/s10836-014-5438-8

Developing Inherently Resilient Software Against Soft-Errors Based on Algorithm Level Inherent Features

Bahman Arasteh · Seyed Ghassem Miremadi · Amir Masoud Rahmani

Received: 8 September 2013 / Accepted: 3 February 2014 / Published online: 25 February 2014

Abstract A potential peculiarity of software systems is that a large number of soft-errors are inherently derated (masked) at the software level. The rate of error-deration may depend on the type of algorithms and data structures used in the software. This paper investigates the effects of the underlying algorithms of programs on the rate of error-deration. Eight different benchmark programs were used in the study; each of them was implemented by four different algorithms, i.e. divide-and-conquer, dynamic, backtracking and branch-and-bound. About 10,000 errors were injected into each program in order to quantify and analyze the error-derating capabilities of different algorithm-designing-techniques. The results reveal that about 40.0 % of errors in the dynamic algorithm are derated; this figure for backtracking, branch-and-bound and divide-and-conquer algorithms are 39.5 %, 38.1 % and 28.8 %, respectively. These results can enable software designers and programmers to select the most efficient algorithms for developing inherently resilient programs. Furthermore, an analytical examination of the results using one-way ANOVA acknowledged the statistical significance of difference between the algorithm-designing-techniques in terms of resiliency at 95 % level of confidence.

Responsible Editor: M. Sonza Reorda

B. Arasteh (✉) · A. M. Rahmani
Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran
e-mail: b_arasteh2001@yahoo.com

A. M. Rahmani
e-mail: Rahmani@srbiau.ac.ir

S. G. Miremadi
Dependable Systems Laboratory, Department of Computer, Sharif University of Technology, Tehran, Iran
e-mail: miremadi@sharif.edu

Keywords Soft-error · Error-derating · Algorithm-designing technique · Resilient program · Fault injection

1 Introduction

Software systems and their applications are deemed to be an omnipresent feature of modern life. One important area of software applications is safety-critical applications, where failures may threaten users or bring about disastrous amounts of monetary and human loss. As a matter of fact, the reliability is judged to be a pivotal and fundamental issue to be discussed in the realm of safety-critical systems.

The studies in [37, 40, 50–52] indicate that soft-errors are regarded as one major source of failures in computer systems. *Alpha particles*, *energetic neutrons* and *protons* caused by cosmic rays and *thermal neutrons* are the main sources of soft errors [37, 40]. The significant reduction in the size and voltage of transistors increases the density and clock speed of processors; as a result, the probability of soft-error occurrence rises in computer systems.

Performance overhead is a serious problem in software-based techniques proposed for detecting and recovering soft-errors [8, 9, 18, 20, 24–30, 39]. Duplicated instructions and consistency checking are the main sources of introduced performance-overhead in the software-based methods. The introduced performance-overhead may result in timing failures in safety-critical and real-time applications.

An interesting and potential peculiarity of computer systems is that a significant number of soft-errors (close to 40 %) are inherently masked at the software levels due to properties of software [6, 17, 19, 21, 33, 47, 49]. A soft error is said to be *derated* if it is inherently masked in the system before it affects the results. A program is deemed to be more *resilient* if an occurred soft-error is more likely to be derated inherently and hence, it can generate correct results [6, 17, 33, 47, 48].

Different implementations of a program often have different impacts on its resiliency [6, 17, 33, 47]. Hence, the inherent resiliency of a program against soft-errors can be considered as a function of the corresponding *algorithms*, *data structures* and *coding* features.

The underlying algorithmic features of a program[1] may have a considerable impact on its derating-rate. The investigation of the effects of different algorithm-designing-techniques on the rate of error-deration is the major purpose of this study. In this regard, the two following questions should be answered:

1. Which structural features of algorithm-designing techniques may increase the error-derating rate?
2. Which algorithm-designing-techniques may have higher error-derating capabilities?

Based on the two above questions, the main contributions of this study are as follows:

1. Structural features of algorithm-designing-techniques affecting error-deration rate are identified.
2. Error-derating capabilities of different algorithm-designing-techniques with respect to their structural features are investigated.
3. Based on the numbers 1 and 2, an approach is proposed to identify the error-derating blocks of a program with regard to the underlying algorithm.
4. A software-based experimental framework for evaluating the reliability of software is developed.

The rest of the paper is organized as follows: Section 2 gives a background and explains terminologies related to the error-deration at algorithm level. Section 3 describes the error model, error-injection tools, benchmarks and the details of the experiments. Section 4 includes the following points: 1) the algorithmic features which may affect error-deration rate are discussed, 2) the error-derating capabilities of different algorithm-designing-techniques are investigated, 3) an approach for the identification of the error-derating blocks of a program is proposed and 4) the ANOVA (analysis of variance) is used to analyze the statistical significance of difference among algorithm-designing-techniques in terms of resiliency. Finally, Section 5 concludes the paper.

2 Background

Duplicating program instructions and comparing their results is a traditional approach to detect soft-errors [3, 8, 18, 20, 24–26, 28, 39]. This approach imposes a considerable performance-overhead to the system [8, 9, 18, 20, 24–26]. To make a trade-off between reliability and performance-overhead in the software-based techniques, the selective duplication instead of full duplication is introduced [3–5, 10, 12, 14, 34, 38, 42, 44]. As a result, it is possible to obtain high reliability with a low performance-overhead.

An interesting feature of computer systems is that a considerable number of soft-errors may be derated inherently in different layers of computer systems [6, 17, 21, 33, 46, 47, 49]. A significant number of soft-errors may be derated inherently at the logical and micro-architectural levels [15, 46, 51, 52]. For instance, a soft-error will be derated logically if it changes an input of AND gate while the other input is zero; consequently, the soft-error will not propagate to the output [15]. On the other hand, a soft-error will be derated electrically when it is attenuated by subsequent logic gates due to the electrical features [15, 52]. Furthermore, a soft-error will be derated if it changes the input of a gate during the time the gate is inactive.

Unmasked and undetected soft-errors at the micro-architectural level will propagate to the higher levels of the system and will become visible at the software level. The results of some studies [6, 15, 17, 19, 21, 33, 47, 49] indicate that a considerable number of soft-errors (close to 40 %) are inherently derated at the program level. Furthermore, programs with different implementations have different derating rates [6, 33, 47]. It can be argued that programs with the error-derating potentials are inherently more resilient against soft-errors. Thus, improving the inherent resiliency of programs without using external redundancy motivates researchers to study the potential sources of error-deration at program level. To the best of authors' knowledge, there are very few studies on soft-error deration at the program level.

Based on the results of experiments in [17, 32], only a small portion of program instructions have a major impact on the final results and produces the same overall results as all instructions produce; the remaining instructions are considered as ineffectual instructions. The ineffectual instructions such as *dynamically useless-codes*, *quality related codes* and *equivalent branches* do not precisely relevant to program results [6, 21, 33, 47]. A large number of soft-errors which affect the ineffectual instructions do not propagate to the program results and may be masked inherently [1, 6, 10, 17, 19, 21, 33, 47, 49].

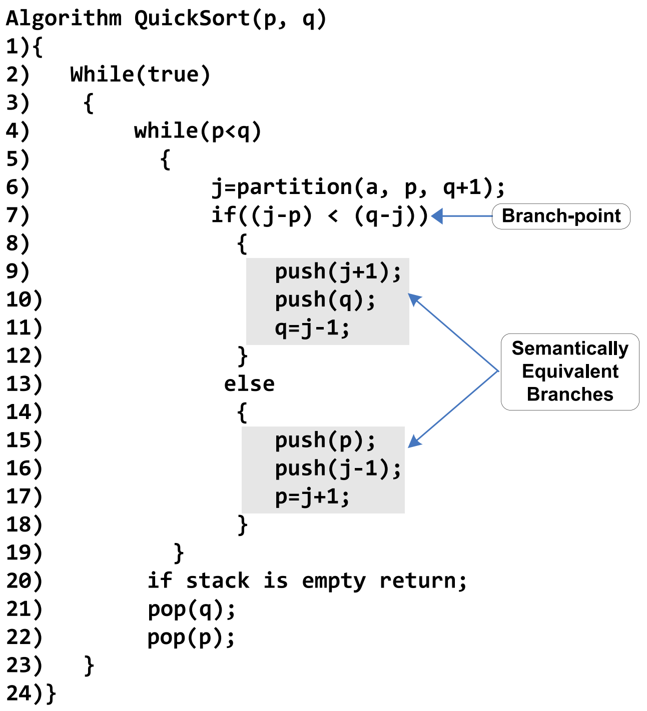
According to the experimental results of some studies [21], on average 14 % of the instructions executed by programs are dynamically useless (dead). The amount of Dynamically Useless-Codes (DUC) in a program may vary based on the underlying algorithms and data structures. DUC generate useless values which do not have impact on the program results at specific time-slices. Hence, non-negligible number of soft-errors may be derated by DUC in a program.

[1] The terms "software" and "program" have been used interchangeably with the same meaning in this paper.

The authors in [41] have investigated the sources of inherent error-masking at different layers of the system to study hardware and program vulnerability. According to the results of the experiments reported in [41], approximately 57 % of masked error at the program level are attributed to dead instructions and 42 % to logical masking. A methodology was proposed in [35] to estimate the reliability of computer systems against soft-errors and it was compared with the traditional methods of fault-injection; in this study [35], the instruction-sets of two microprocessors (8088 and OR1200) were examined and the rate of errors masked by these instructions were quantified. However, these studies did not investigate the impact of different programming and algorithm-designing techniques on the rate of error-deration; this is deemed to be a research gap in this area of study.

Figure 1 depicts an iterative algorithm for the *quick-sort* [13]. The conditional branch in line 7 is the branch point of equivalent branches. The deviation of this branch direction which is caused by soft-errors does not have impact on the final results; hence, the soft-errors will be derated at the program level. Indeed, this algorithm is inherently more resilient to soft-errors.

In this study, the effects of algorithm-designing techniques on the rate of error-deration will be investigated. To this end, firstly, the dynamic behaviors and the structures of different algorithms will be analyzed to identify those algorithmic features which result in the DUC, equivalent branches and quality-related codes; these algorithmic features are considered as potential sources of error-deration in an algorithm. Secondly, error derating capabilities of different algorithm-designing techniques will be investigated with regard to the sources of error-deration. To achieve this purpose, a series of error-injection experiments were performed. The following section describes the details of the experimental framework.



```
Algorithm QuickSort(p, q)
1){
2)   While(true)
3)    {
4)        while(p<q)
5)          {
6)              j=partition(a, p, q+1);
7)              if((j-p) < (q-j))
8)                {
9)                   push(j+1);
10)                  push(q);
11)                  q=j-1;
12)               }
13)              else
14)               {
15)                  push(p);
16)                  push(j-1);
17)                  p=j+1;
18)               }
19)          }
20)        if stack is empty return;
21)        pop(q);
22)        pop(p);
23)   }
24)}
```

Fig. 1 An Iterative version of quick-sort algorithm [13] and its equivalent branches in the *dark text-boxes*

3 Experimental Framework

To study algorithm-level error-deration, we conducted a series of error-injection experiments on different benchmark programs using different algorithms. The behaviors of erroneous programs were examined in the presence of the injected errors to investigate the effects of different algorithms on the rate of error-deration. The error model in our experiments is explained in Subsection 3.1. Then, the simulation framework and the features of the benchmarks are described in Subsection 3.2.

Next, the details of error-injection experiments are illustrated in Subsection 3.3.

3.1 Error Model

In this paper, the term *soft-error* refers to a bit-flip which is visible to the program. Indeed, the effect of soft-error in the memory is modeled by inverting a bit in randomly selected memory locations (code, data and stack segments) at a randomly selected clock cycle. A single-bit flip leads to the production of a code-error or data-error.

In this study, only the errors which are visible to the program have been taken into account and their effects on the program output are examined. The injected errors target the code and data of a program regardless of the locations of the injections. In the main memory, the corrupted code and data will ultimately be fetched to the cache and internal registers of the microprocessor during program execution.

In [7], it was shown that approximately 99 % of errors in the 180 nm technology are single-bit errors. This figure for the 40 nm technology is about 60 %. Whereas the rate of multiple-bit errors increases in novel technologies, the rate of single bit errors is still a main concern in all technologies. As such, single-bit errors were investigated in this study. It should be maintained that studying the effect of multiple-bit errors is recommended as a direction for future study (see the list of future works).

3.2 Benchmarks

The SimpleScalar tool set was used in our experimental framework [2]. It includes *sim-safe* and *sim-outorder*. The SimpleScalar, as a functional simulator, is used for simulating the architecture of a processor. The SimpleScalar is distributed with self-tests to validate its results and includes standard libraries and modules. The *sim-safe* simulator is used to run the benchmarks and to evaluate the rate of error-deration. The configuration parameters of the simulator are given in Table 1.

Our experiments require such a set of benchmark programs which illustrate the behaviors of different algorithm-designing-techniques; this is why large benchmark suites such as SPEC2006 [11] were not used in the study. Hence, eight different benchmarks were selected and each benchmark was implemented by four different algorithms as a version of the benchmark. Each version of benchmarks is related to an algorithm-designing-technique. The selected set of benchmarks consists of numeric, nonnumeric, combinatorial and graph-related programs. Thirty two benchmarks which were implemented in the study were classified into four categories based on their underlying algorithms:

- Divide-and-conquer based benchmarks
- Dynamic based benchmarks
- Backtracking based benchmarks
- Branch-and-bound based benchmarks

Each of the above-mentioned algorithm-designing techniques will be described in detail in Subsection 4.2. The benchmarks shown in Table 2 were written in the C programming language and were compiled on Linux (Ubuntu 9.10) using GCC 4.0.2 with standard optimization level (−O2).

In Table 2, all the v1 columns represent the first versions of the benchmarks which use divide-and-conquer algorithm. All the v2 columns stand for the second versions which use dynamic algorithm. All the v3 columns indicate the third versions which are based on backtracking algorithm and finally all v4 columns are related to the forth versions which use branch-and-bound algorithm. Each category of the benchmarks represents the behavior of an algorithm-designing technique. They have been used to investigate the resiliency of underlying algorithms. The errors in the header files and dynamically linked functions are not considered in the experiments. The results of error-injection experiments are classified and shown in Table 3.

Table 1 Configuration of SimpleScalar simulator

Processor parameters	Value
Instruction fetch queue size (if qsize)	4
Branch predictor type	bimod
Register update unit (RUU) size	16
load/store queue (LSQ) size	8
Total number of integer ALU's available	4
Total number of integer Multiplier/divider	1
inst/data TLB miss latency (in cycles)	30
Register File	32 INT and 32 FP
Memory hierarchy parameters	Value
Memory access bus width (in bytes)	8
Instruction L1	16 KB
Data L1	16 KB
Base address of code segment	0x00400000H
Base address of initialized data segment	0x10000000H
Base address of stack segment	0x7fffc000H

3.3 Error Injection

In order to investigate the effects of different algorithms on the rate of error-deration, we carried out error-injection experiment in the memory hierarchy of each benchmark. After the injection of each error, the behaviors of erroneous programs were monitored and were compared with a golden run to determine the effects of the errors.

Any error-injection experiment should target all possible locations (code and data) of each benchmark at different clock-cycles. Furthermore, the number of injected errors in each benchmark has a significant impact on the accuracy of the experimental results. Hence, the number of injections should be determined based on the following characteristics of the benchmarks: size of code and data and rates of integer and branch instructions. In this study, the number of injections was set at 10,000 for each benchmark with respect to the characteristics of the most complex benchmarks, namely *TSP* and *Knap*. In each experiment, only one bit was flipped in a randomly selected memory location at a randomly selected clock-cycle.

In order to illustrate the adequacy of 10,000 injections for our benchmarks, the researchers prepared two steady-state diagrams. In this study, the *steady* state refers to the state in which the resiliency of a program does not change remarkably though we change the number of injections. That is, in a steady state, increasing the number of injections has no considerable effect on the resiliency of a program. Thus, it should be noted that the number of injections achieving the steady state is regarded as adequate. In order to specify the adequate number of injections, the authors conducted a series of error injection experiments on *TSP* and *Knap*. In our experiments, a range of errors (1–10,000) were injected in both benchmarks to determine the steady state.

As shown in Figs. 2 and 3, when the number of injections reaches 10,000 both benchmarks achieve the steady state. Hence, when the number of injections was set at 10,000, all codes and data in each benchmark were targeted by the errors at different clock-cycles. Table 4 presents standard deviation as an index of dispersion for different numbers of injections. The obtained standard deviations when 1–4,000 errors were injected in *TSP* and *Knap* benchmarks were 9.23 and 10.34 respectively. However, it was observed that when 4,000–8,000 errors were injected, the standard deviations were reduced to 2.74 and 2.71 respectively. Furthermore, when

Table 2 Characteristics of the used benchmarks in the experiments

Benchmark	Description	Complexity class	Problem type	Input variable		Num. of instructions committed for total runs (in billions)				Rate of integer computations (%)			
				Description	Range	V1	V2	V3	V4	V1	V2	V3	V4
Fib	Computing the nth element of the Fibonacci sequence	P	Numerical problem	Fibonacci (n)	$18<n<30$	0.12	0.0002	0.0005	0.0005	44.1	54.9	59.1	58.4
Bin	Computing binomial coefficient for values n, k.	P	Numerical problem	Bin (n, k)	$2<k<30$ $20<n<70$	89.91	41.10	50.70	48.30	33.6	9.1	18.2	19.2
Pow	Computing x^n	P	Numerical problem	Pow (x, n)	$1<x<10$ $2<n<30$	27.49	27.48	27.50	27.49	31.9	32.1	31.7	31.7
TSP	Solving the classic TSP problem [13, 22, 45]	NP-Complete	Combinatorial problem (graph problem)	n: num. of graph node $W_{i,j}$: weight of $\text{edges}_{i,j}$ (Rnd)[a]	$5<n<25$ $0<W_{i,j}<30$	58.27	85.06	29.47	31.30	11.0	11.3	2.1	6.8
Rod-c	Solving the classic rod-cutting problem [22, 45]	P	Combinatorial problem	n : length of rod P_i : price of piece_i (Rnd)	$10<n<30$ $1<P_i<40$	57.86	1.85	5.21	4.82	35.2	61.7	48.7	49.3
Knap	Solving the classic 0–1 knapsack problem [13, 45]	NP-Complete	Combinatorial problem	n : capacity P_i: profit of item_i (Rnd) W_i: weight of item_i (Rnd)	$5<n<55$ $1<P_i<20$ $1<W_i<20$	819.14	816.88	81.09	92.91	8.8	61.1	9.8	5.8
Queen	Solving the classic n-queen problem [13, 22, 45]	NP-Complete	Combinatorial problem	n: num. of queens	$4<n<16$	2.02	2.01	1.70	1.81	55.7	56.1	58.8	58.3
S_Sub	Solving the classic subset sum problem [13, 22, 45]	NP-Complete	Combinatorial problem (set problem)	n: num. of items W_i : weight of item_i (Rnd) Sum: required sum	$5<n<40$ $1<W_i<25$ $5<\text{Sum}<30$	2.91	2.31	1.62	1.90	12.1	53.2	24.1	15.0

[a] Rnd refers to the randomly generated numbers under uniform distribution

Table 3 Classification of the results for error-injection experiments

Result classes	Description	Detection mechanism
Correct	Production of correct output by the program	Checking result with golden run
Fail-Silent Data Corruption (SDC)	Production of incorrect output by the program	Checking result with golden run
Exception	Hang: program timeout	Checking the benchmark execution time with time of golden run
	Crash: Abnormal program termination (invalid instruction, invalid memory address, overflow, segmentation error)	Detection by the hardware and software exceptions

8,000–10,000 errors were injected, the value of standard deviation was minimized.

Hence, it should be argued that when the number of injections reaches 10,000, the dispersion and variability of the results as measured by standard deviation is significantly reduced. That is, 10,000 injections are adequate for studying the resiliency of the mentioned benchmarks.

In each experiment the results of erroneous program and golden runs have been compared and saved in a file as the report of the experiment. Each report includes the number of correct, SDC and exception for each experiment. In total, about 320,000 experiments were conducted on a computer equipped with an Intel 2.8 GHz quad-core processor and 4GB of RAM. All the benchmarks do not include the I/O instructions and their input data have been predefined and stored. The basic presumptions of the experiments are as follows:

- Probability distribution for the input data is assumed to be uniform.
- Flipping a bit in the code segment changes an op-code or an operand (immediate or address operand) but flipping a bit in the data segment changes the value of a data in the program. An injected error is visible at the program level.
- The time at which an error is injected during the program execution is determined randomly.
- An error (a single bit flip) is dynamically injected into the randomly selected memory points in a uniform distribution.

4 Experimental Results

This section describes the results of the experiments as follows: 1) the potential sources of error-deration at the algorithm level are discussed, 2) the identified sources of error-deration are utilized to investigate the error-derating capabilities of different algorithm-designing techniques, 3) based on results 1 and 2, an approach is proposed to identify the vulnerable blocks of a program with regards to its underlying algorithm, and 4) one-way ANOVA [31] is used to statistically analyze the significance of difference among algorithm-designing techniques in terms of error-derating rates at 95 % degree of confidence.

4.1 Sources of Error-Deration

In this subsection, the sources of error-deration in the algorithm-designing techniques are examined. To this end, dynamic behaviors of different algorithms are analyzed in the presence of the injected errors; then, structures of different algorithms are examined to find out the potential sources of

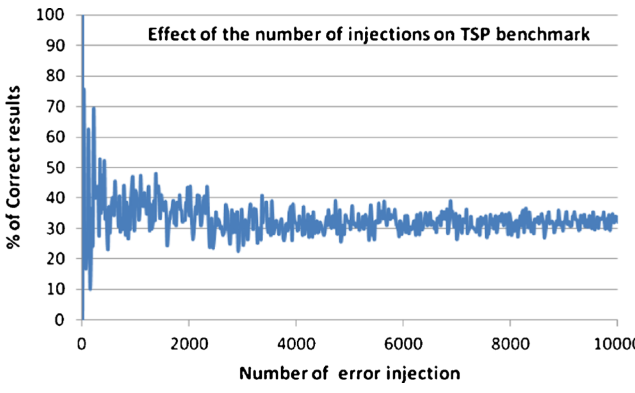


Fig. 2 Steady-state diagram illustrates the adequacy of 10,000 injections in TSP benchmark

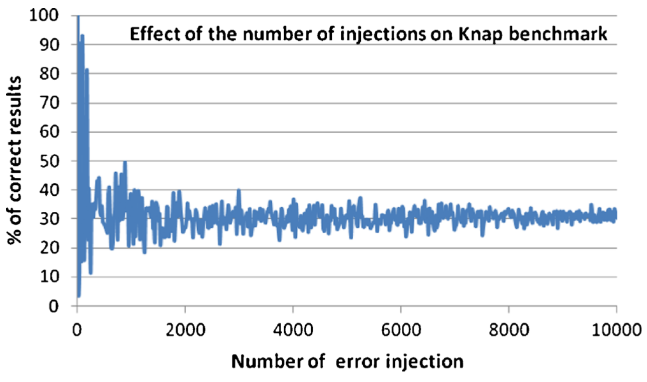


Fig. 3 Steady-state diagram illustrates the adequacy of 10,000 injections in *Knap* benchmark

Table 4 Standard deviation for different numbers of injections

	Number of error-injections		
	1–4,000	4,000–8,000	8,000–10,000
STD. Deviation in TSP	9.23	2.74	1.98
STD. Deviation in Knap	10.34	2.71	1.64

error-deration. Indeed, the error-injection experiments are carried out and the effects of errors on the behaviors of different benchmarks are studied. Figure 4 demonstrates the results of our error-injection experiments. The average resiliency of all benchmarks in the experiments indicates that approximately 37 % of the injected errors are derated by the software without altering the results.

One of the interesting points is that different versions of each benchmark have different resiliency. For example, the amounts of resiliency in the different versions of *n-Queen* program are respectively 30 %, 37 %, 44 % and 41 %. Indeed, different implementations of a program have different derating-rate. According to the results of the experiments, the second version of each benchmark has the highest resiliency among all versions where in the first version the resiliency is low. For example, the resiliency in the first version of *knap* program is 22 % whereas the resiliency in the second version is 31 %.

As mentioned in Section 3, all versions of each benchmark were written in the same programming language and were compiled by the same compiler at the same optimization level. However, since the only difference among benchmarks is their algorithms, hence, the algorithmic features are analyzed to discover the sources of error-deration. The memoization technique [13, 22, 45] is used in the second versions of all the benchmarks. Memoization is considered as a classic technique which can be used in different algorithms for performance enhancement. Also, the recursive divide-and-conquer [13, 22, 45] as a classic and efficient technique, is used in the first versions of all benchmarks. Pruning and bounding techniques [13, 22, 45] are used respectively in the third and fourth versions of the benchmarks.

With respect to the requirements of software, these well-known techniques (memoization, pruning and bounding) can be used only for enhancing the performance of the software. Until now, negligible concern has been given to the potential effects of these techniques on the reliability. As demonstrated in Fig. 4, the results of error-injection experiments and the analysis of different algorithms indicate that these techniques have unexpected and potential impacts on the resiliency of an algorithm; that is to say, the following techniques at the algorithm-level can be considered as the potential sources of error-deration:

- Memoization
- Pruning
- Bounding

These techniques are discussed one by one in the following part.

Memoization: Memoization is a well-known optimization technique for speeding up algorithms [13, 22, 45]. Storing computed results to avoid recomputation is the main purpose of memoization. Figure 5 illustrates an

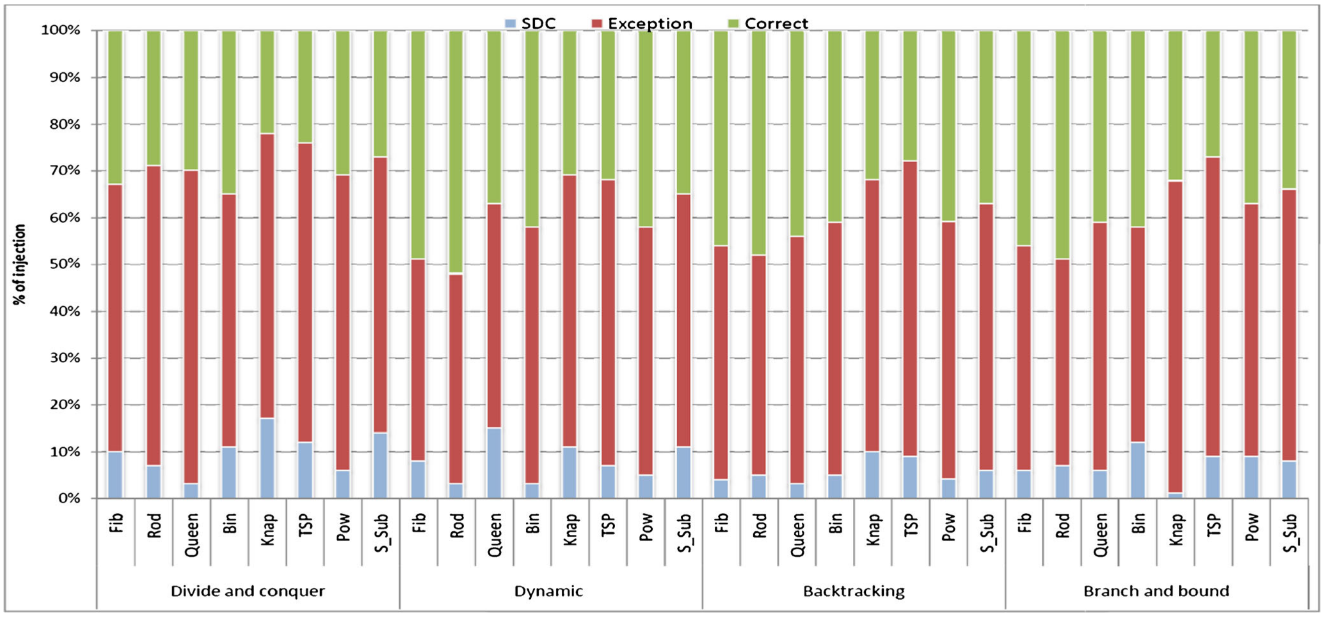
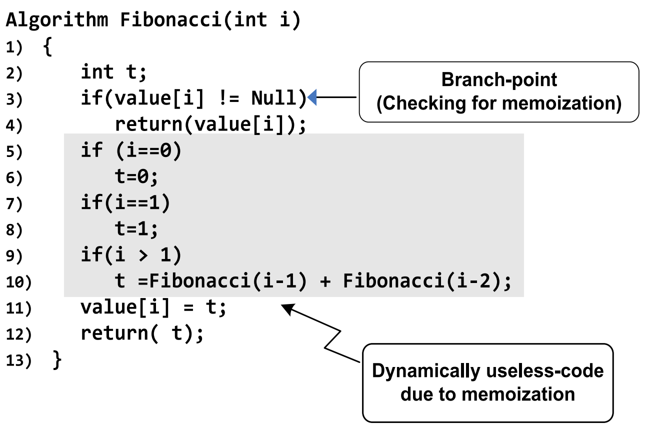


Fig. 4 Average results of error injections in different benchmarks



```
Algorithm Fibonacci(int i)
1)  {
2)     int t;
3)     if(value[i] != Null)
4)        return(value[i]);
5)     if (i==0)
6)        t=0;
7)     if(i==1)
8)        t=1;
9)     if(i > 1)
10)       t =Fibonacci(i-1) + Fibonacci(i-2);
11)    value[i] = t;
12)    return( t);
13) }
```

Fig. 5 An algorithm for calculating Fibonacci series [36]. The codes in the dark text-box are DUC caused by memoization

algorithm for calculating the *Fibonacci* series which uses the memoization technique; this algorithm has been discussed in algorithm related books [13, 22, 36, 45]. This algorithm is equipped with memoization technique to save the computed value and to avoid recomputation. Avoiding any recomputation in an algorithm leads to a drastic reduction in the number of recursive calls.

After analyzing the static and dynamic behaviors of the benchmarks using memoization, we found that memoization results in the inherent and implicit production of the following features:

- equivalent branches
- dynamic useless-code (DUC)

In the memoization technique, a branch instruction is used to avoid recomputations and improve the performance (line 3 in Fig. 5). Such an instruction might have no significant impact on the final results. That is to say, the performance-related instructions may derate soft-errors.

The *sim-profile* was used for profiling the benchmarks [2]; the results of program profiling show that approximately 40 % of corresponding machine codes of the *Fibonacci* program are DUC at the run time. In the *Fibonacci* program, DUC is only executed at about 60 % of the total run-time and in the remaining run-time, it is useless. Hence, the injected errors into DUC at 40 % of run-time will be derated.

To sum it up, it can be maintained that the memoization technique in an algorithm inherently produces equivalent branches and DUC; consequently, it can implicitly enhance the resiliency of the corresponding algorithm. For example, as the results of the experiments indicate, the average resiliency of *Fibonacci* program using memoization is 49 % whereas its resiliency without using memoization is 33 %. Thus, in addition to the main purpose of this technique which is performance enhancement, memoization should also be considered as one of the inherent and potential sources of error-deration at the algorithm level. Figure 6 illustrates the effect of memoization technique on the resiliency of different programs in the presence of the injected errors.

Using memoization in an algorithm will inherently improve the resiliency against the soft-errors. The memoization needs a memory block for storing pre-calculated values. This block does not include controlling data but includes only the pre-calculated values which might be used in the future computations. The analysis of benchmark behaviors, namely *Fib* and *Pow*, illustrates that only two memory words are required for storing the computed values. That is to say, in case a large number of pre-calculated values are frequently reused, those values have to be protected for preserving the memoization resiliency. The frequently used values in the memory can be protected with ECC.

Pruning: Programmers and software designers sometimes deal with optimization problems and the algorithm proposed for such a problem must, at the worst case, traverse the solution tree (possible solutions) from the root [13, 22, 45]. The pruning as a well-known and performance enhancing technique is aimed to reduce the search space (solution tree) and consequently reduce the time required for the computations in an algorithm. Indeed, the solution tree is pruned so as to avoid nodes that are not promising. During traversing the solution tree a node is assumed none promising if it cannot possibly lead to a solution [13, 22, 45]. Figure 7 illustrates an algorithm for the *n-Queen* problem as a classic NP-complete problem [13, 22, 36, 45]. This algorithm locates n Queens on an $n \times n$ chessboard so that no two Queens threaten each other. The array *column* in this algorithm determines the locations of n Queens. In this algorithm the branch instruction in line 2 is used to prune out the inconsistent locations.

Having analyzed the behaviors of the benchmarks using the pruning, we noted that applying this technique

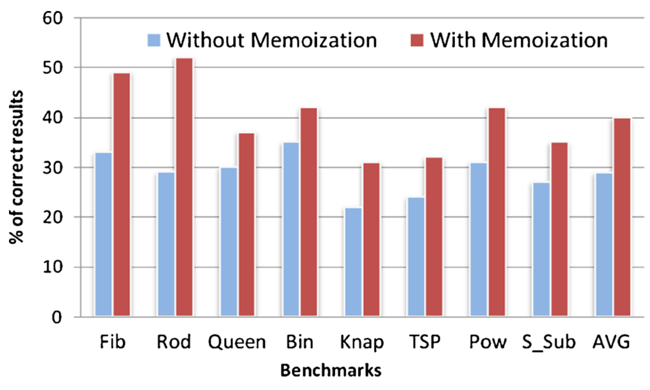
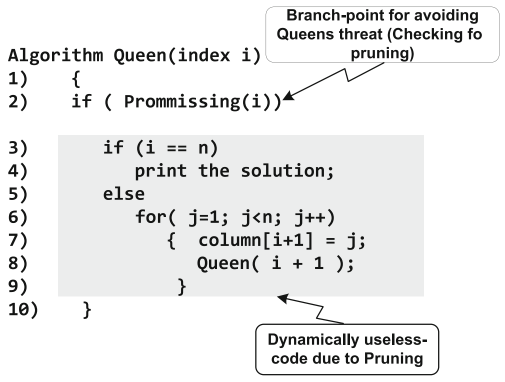


Fig. 6 The results of error injection experiments in the different programs with and without memoization


Branch-point for avoiding Queens threat (Checking fo pruning)
Dynamically useless-code due to Pruning

```
Algorithm Queen(index i)
1)    {
2)    if ( Prommissing(i))

3)       if (i == n)
4)          print the solution;
5)       else
6)          for( j=1; j<n; j++)
7)             {  column[i+1] = j;
8)                Queen( i + 1 );
9)             }
10)    }
```

Fig. 7 An algorithm for locating n Queens on an $n \times n$ chessboard [13, 22, 36, 45]

to an algorithm leads to the inherent and potential production of the following features:

- equivalent branches
- Dynamically Useless-Code (DUC)

  In general, a conditional branch instruction in the pruning technique is intended to determine the promising nodes in the solution tree (second line of Fig. 7); indeed, this branch instruction is a performance-related branch and, any deviation from its correct direction by errors may have no impact on the results. Hence, this branch is less vulnerable to soft-errors.

  The other notable point is that pruning technique leads to the inherent and implicit production of DUC (codes in the dark text-box in Fig. 7). According to the profile of the *n-Queen* program, the codes in the dark text-box of Fig. 7 are useless in about 60 % of calls. Hence, the presence of DUC enhances the resiliency of the respective algorithm.

  In brief, it can be argued that the pruning technique in an algorithm inherently produces equivalent branches and DUC; as a result, it potentially enhances the resiliency of the corresponding algorithm. As the results of the experiments reveal, the average resiliency of the *n-Queen* program using pruning is 44 % whereas its resiliency without pruning is 30 %. Thus, pruning technique should be considered as one of the inherent and potential sources of error-deration although its main function is performance enhancement. Figure 8 illustrates the effect of pruning on the resiliency of different benchmarks.

Bounding: Bounding can be used in different algorithms to enhance the performance [13, 22, 45]. The bounding technique is in charge of calculating upper and lower bounds to determine which nodes of the solution tree are ineffective and must be discarded from the range [13, 22, 45].

Analyzing the behavior of different benchmarks using the bounding technique reveal that the inclusion of this

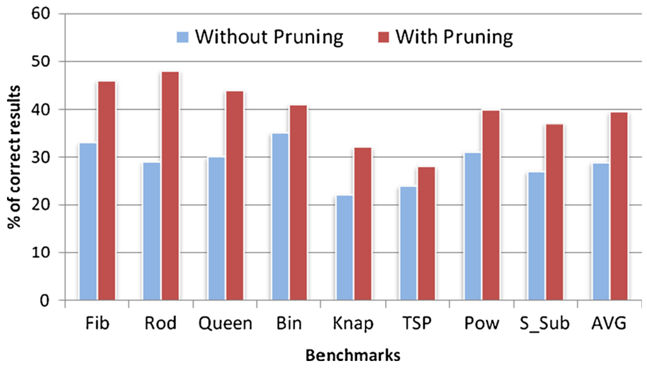


Fig. 8 The results of error injection experiments in different programs with and without pruning

technique in a program potentially leads to the production of the following features:

- equivalent branches
- Dynamically Useless-Code (DUC)

  The bounding technique in an algorithm inherently produces equivalent branches and DUC; hence, it can potentially enhance the resiliency of the corresponding algorithm besides enhancing the performance. As the results of the experiments show, the resiliency of *0–1 knapsack* program using bounding is about 30 % whereas its resiliency without using bounding is about 22 %.

  Based on the results of the experiments, the authors found that memoization, pruning and bounding can be regarded as the potential sources of error-deration in different algorithms. These sources are utilized for investigating the error-derating capabilities of different algorithm-designing techniques.

4.2 Error-Derating Capabilities of Algorithm-Designing Techniques

Over the years, software developers have used different well-known techniques such as *divide-and-conquer* and *Dynamic* to design effective algorithms for solving a wide range of problems. It should be maintained that the fitness of each technique depends upon features of the given problem. For example, the *matrix-multiplication* problem can be solved efficiently by the divide-and-conquer technique. As another example, the efficient algorithm for the *0–1 Knapsack* problem was devised based on the divide-and-conquer and dynamic techniques [13, 22]. In this subsection, the error-derating capabilities of different algorithm-designing techniques are investigated. The followings are taken into account in the investigation:

- Memoization, branching and bounding are the potential sources of error-deration which improve the resiliency of an algorithm.

- Rates of branch instructions, load/store instructions and system calls (more vulnerable instructions) have impact on the resiliency of an algorithm [5, 10, 44].

In other words, while investigating the resiliency of an algorithm, the sources of error-deration and the rate of vulnerable instructions are taken into account. This study has zoomed in on the widely used following algorithm-designing techniques:

- divide-and-conquer
- dynamic
- backtracking
- branch-and-bound

The derating capability of each technique is explained below.

i. Divide-and-Conquer Algorithms

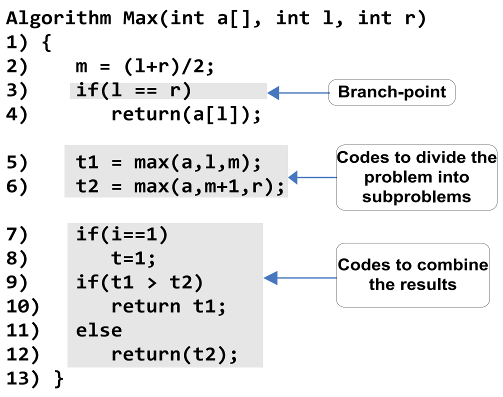
The divide-and-conquer is a well-known and powerful technique for designing efficient algorithms to solve a wide range of problems. As a case in point, this technique has an outstanding role in the efficiency of algorithms such as *quick-sort*, *merge-sort*, *strassen* algorithm for matrix multiplication and f*ast-Fourier* transform algorithm [13, 16, 22, 36, 45]. A divide-and-conquer algorithm divides an instance of a problem with the size *n* into several smaller sub problems [13, 22, 36, 45]. The solutions to the sub-problems are combined into an overall solution to the entire problem. The divide-and-conquer paradigm includes *divide, conquer* and *combine* as three steps at each level of the recursion. Figure 9 shows a divide-and-conquer algorithm for finding the maximum element of a list [13, 22, 36, 45].

Structural Feature: s In general, the recursively-implemented divide-and-conquer algorithms include dividing and combining operations. Figure 10 illustrates a general control-flow graph (CFG) of the divide-and-conquer algorithms. Analyzing the structures of divide-and-conquer algorithms reveals that these algorithms lack potential sources of error-deration (memoization, pruning and bounding). One more inherent feature of divide-and-conquer algorithms is that they do not generate DUC.

Rate of Vulnerable Instructions: As mentioned earlier, an increase in the rate of branch instructions, system calls and load/store instructions will raise the vulnerability of the program against soft-errors [5, 10, 44]; consequently, the rate of these vulnerable instructions should be taken into account when investigating the derating capability. As a profiling tool, the *sim-profile* [2, 43] is used to gather data on the dynamic behavior and vulnerable instructions of each algorithm-designing technique. The gathered data in our profiling experiments include the instruction profile, rate of branches, rate of system-calls, rate of load/store instructions, address profile and page-table access profiles.

Figure 11 illustrates the rate of branch instructions in different benchmarks using different algorithms. On average, the rate of branch instruction in divide-and-conquer, dynamic, backtracking and branch-and-bound are respectively 32.77 %, 21.71 %, 33.65 % and 32.81 %. The bars in Fig. 12 illustrate the average number of instructions per branch (IPB) in each benchmark. IPB is computed by dividing the total number of executed instructions by the number of BBs. Unlike the other factors, an increase in the rate of IPB increases the resiliency of the corresponding program.

As another case, the rate of load/store instructions in the divide-and-conquer algorithms is 36.42 %



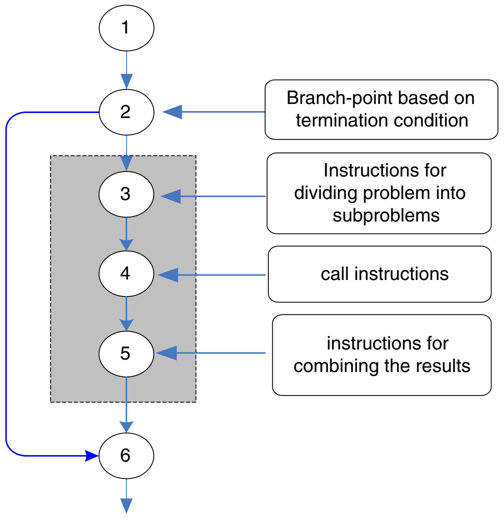
```
Algorithm Max(int a[], int l, int r)
1) {
2)     m = (l+r)/2;
3)     if(l == r)
4)        return(a[l]);

5)     t1 = max(a,l,m);
6)     t2 = max(a,m+1,r);

7)     if(i==1)
8)        t=1;
9)     if(t1 > t2)
10)       return t1;
11)    else
12)       return(t2);
13) }
```

Branch-point
Codes to divide the problem into subproblems
Codes to combine the results

Fig. 9 A divide-and-conquer algorithm for finding maximum element of a list [13, 22, 36, 45]



1
2
3
4
5
6
Branch-point based on termination condition
Instructions for dividing problem into subproblems
call instructions
instructions for combining the results

Fig. 10 A general form of CFG for divide-and-conquer algorithms

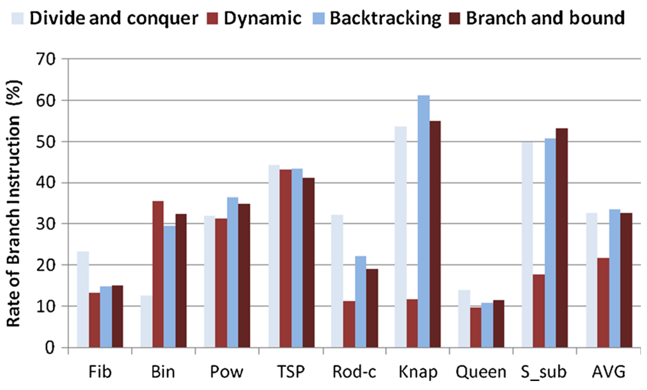


Fig. 11 The rate of branch instructions in different benchmarks

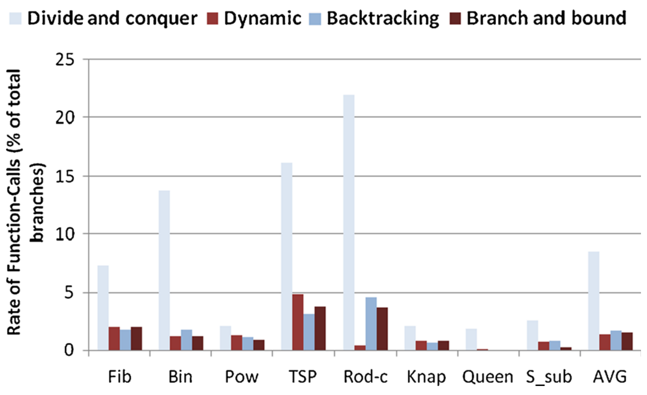


Fig. 13 The rate of function-calls in different benchmarks

whereas the average rates for the same instructions in the other algorithms is 33.17 %. Furthermore, the rate of function-calls in divide-and-conquer is 8.48 % whereas the average rate of function-calls in the other algorithms is 1.58 %. Figure 13 shows the rate of function-calls in different benchmarks using different algorithms. Figure 14 illustrates the number of system-calls for 10,000 runs of each benchmark. The average numbers of system-calls in divide-and-conquer, dynamic, backtracking and branch-and-bound algorithms are respectively 266.26 k, 7.40 k, 81.92 k and 63.96 k.

Having analyzed the structures and dynamic behaviors of divide-and-conquer algorithms, we found that there are no potential sources of error-deration in these algorithms. The absence of error-derating sources and the high rate of vulnerable instructions in divide-and-conquer algorithms result in lower resiliency than the other algorithms. The results of the experiments, as shown in Fig. 4, indicate that divide-and-conquer algorithms have about 10.33 % lower resiliency than the average resiliency of other algorithms. For example, the resiliency of divide-and-conquer algorithm for *Pow* benchmark is about 8.66 % lower than the average resiliency of other algorithms. However, divide-and-conquer algorithms efficiently solve a wide range of problems. The quantified results of our experiments, shown in Fig. 4, acknowledge low resiliency of divide-and-conquer algorithms.

ii Dynamic Algorithms

Dynamic technique is another widely applicable algorithm-designing technique which has been used successfully for a wide range of problems [22, 45]. Two types of this technique include *bottom-up* and *top-down*. The top-down dynamic technique is equipped with memoization technique which results in a drastic reduction in the number of call instructions. Consequently, the memoization technique reduces the running-time of the related algorithms. Figure 15 depicts a top-down dynamic algorithm for solving *0-1knapsack* problem [22, 45].

Structural Features: Figure 16 depicts a general CFG of dynamic algorithms. Analyzing the structure of dynamic algorithms demonstrates that these algorithms include the memoization technique; this technique is intended to be used only for enhancing the performance. Moreover, as mentioned in Subsection

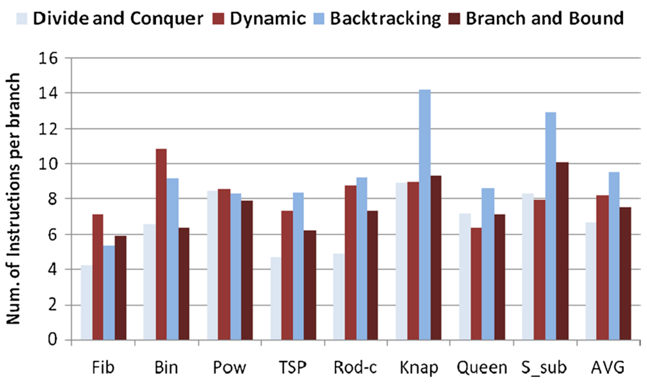


Fig. 12 The average number of instructions per branch (IPB) in each benchmark

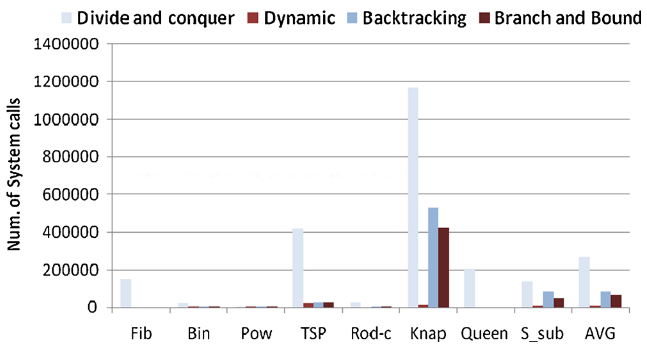
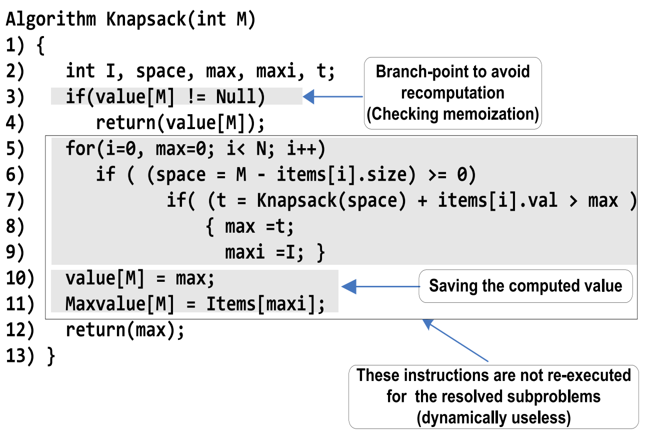


Fig. 14 The average number of system-calls for 10,000 runs of each benchmark



```
Algorithm Knapsack(int M)
1) {
2)    int I, space, max, maxi, t;
3)    if(value[M] != Null)
4)       return(value[M]);
5)    for(i=0, max=0; i< N; i++)
6)       if ( (space = M - items[i].size) >= 0)
7)              if( (t = Knapsack(space) + items[i].val > max )
8)                  { max =t;
9)                    maxi =I; }
10)   value[M] = max;
11)   Maxvalue[M] = Items[maxi];
12)   return(max);
13) }
```

Branch-point to avoid recomputation (Checking memoization)
Saving the computed value
These instructions are not re-executed for the resolved subproblems (dynamically useless)

Fig. 15 A dynamic algorithm for resolving the 0–1 knapsack problem [22, 45]

4.1, the memoization technique can also be considered as a potential source of error-deration. The presence of memoization technique in dynamic-algorithms indirectly results in the production of DUC (block 3 and 4), equivalent branches and multiple-outputs (block 2). As a case in point, regarding the structure of dynamic algorithms the branch instruction in third line of Fig. 15 is a performance-oriented instruction. Furthermore, the instructions in the dark text-box are DUC. Hence, a considerable number of soft-errors which affect DUC and equivalent branches may be derated.

Rate of Vulnerable Instructions: In the benchmarks using dynamic-algorithms, the rate of branch instructions is about 11 % lower than the average rate of the same instruction in other benchmarks. Furthermore, the average number of IPB in dynamic algorithms is equal to 8.22 whereas this factor for divide-and-conquer and branch-and-bound algorithms is respectively 6.64 and 7.53 (shown in Fig. 12). A high rate of IPB in the benchmarks using dynamic algorithm increases the resiliency.

To sum it up, it can be maintained that the presence of memoization technique and the low rate of vulnerable instructions in the dynamic algorithms will result in higher resiliency than those of the other algorithms. The results of the error-injection experiments, shown in Fig. 4, indicate that the average resiliency of dynamic algorithms is about 12 % higher than the average resiliency of divide-and-conquer algorithms.

It should be noted that using the dynamic algorithms for enhancing the resiliency does not have negative effect on the performance. Indeed, these algorithms can efficiently solve a wide range of problems as well as enhancing the resiliency. For example, the average resiliency of dynamic algorithm for the *Fib* benchmarks is about 49 % whereas the resiliency of the corresponding divide-and-conquer algorithm is 33 %. Moreover, the divide-and-conquer algorithm runs in $O(2^n)$ time whereas the dynamic algorithm runs in $O(n)$ time. The results of our experiments in Fig. 4 confirm the high resiliency of these algorithms.

The dynamic algorithms require a memory block to store pre-computed results; this block does not include the controlling data. Analysis of the benchmarks' behavior illustrates that commonly a small part of pre-computed results, stored results, will be used in the next computations during program execution. For example, the dynamic algorithm for computing the Fibonacci series only needs two words of memory locations for storing the pre-computed results. Hence, in the case of programs with a large amount of memoized data, protecting just frequently used data should be taken into account.

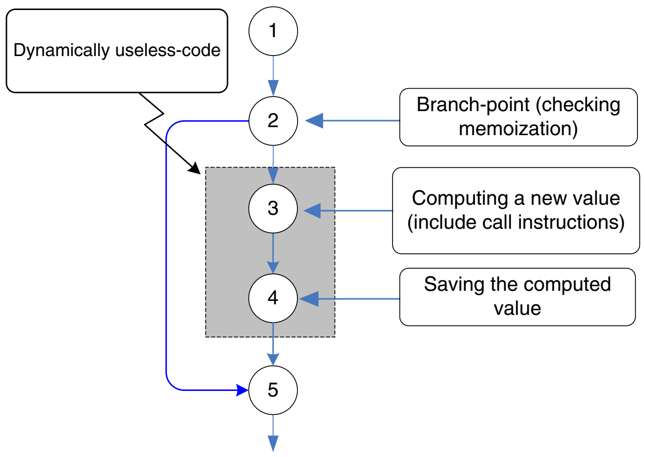
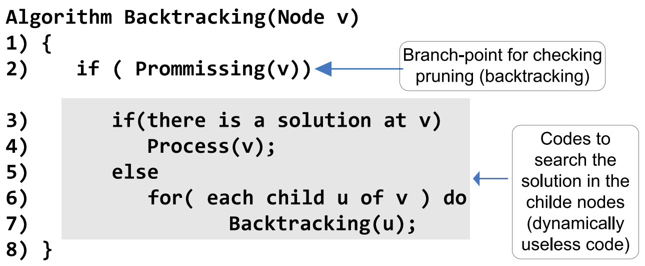

Dynamically useless-code
1
2
3
4
5
Branch-point (checking memoization)
Computing a new value (include call instructions)
Saving the computed value

Fig. 16 The general CFG for dynamic-algorithms. The memoization technique leads to production of equivalent-branches and DUC

iii Backtracking Algorithms

Backtracking is a well-known algorithm-designing technique for solving wide range of problems [13, 22, 45]. Backtracking consists of a depth-first search for a solution and checks whether each node is promising or not. If a node is not promising, it will backtrack the node's parent and continue the search for another child which is called pruning the solution tree. Figure 17 depicts a general structure for the backtracking algorithms.

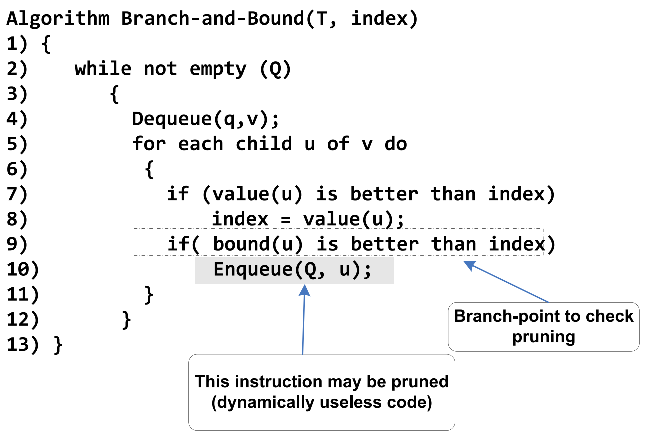
Structural Features: After analyzing the structure of backtracking algorithms, we found that these algorithms include the pruning technique as potential and inherent source of error-deration. The pruning technique is only used to avoid traversing non-



```
Algorithm Backtracking(Node v)
1) {
2)    if ( Prommissing(v))
3)       if(there is a solution at v)
4)          Process(v);
5)       else
6)          for( each child u of v ) do
7)                Backtracking(u);
8) }
```

Branch-point for checking pruning (backtracking)
Codes to search the solution in the childe nodes (dynamically useless code)

Fig. 17 General structure of backtracking algorithms [17, 22, 45]



```
Algorithm Branch-and-Bound(T, index)
1) {
2)    while not empty (Q)
3)       {
4)         Dequeue(q,v);
5)         for each child u of v do
6)          {
7)            if (value(u) is better than index)
8)                index = value(u);
9)            if( bound(u) is better than index)
10)               Enqueue(Q, u);
11)         }
12)      }
13) }
```

Branch-point to check pruning
This instruction may be pruned (dynamically useless code)

Fig. 18 General structure of branch-and-bound algorithms [13, 22, 45]

promising sub-trees and to enhance the performance of the algorithm. As mentioned in Subsection 4.1, the pruning technique is a potential source of error-deration and indirectly produces DUC and equivalent branches. The branch instruction in the second line of Fig. 17 is a performance-related instruction. The instructions in the dark text-box of Fig. 17 are not executed for the non-promising nodes and hence, these instructions are DUC.

Rate of Vulnerable Instructions: With respect to the results of experiments in Fig. 13, the rate of vulnerable instructions such as function-calls in backtracking algorithms is 1.75 % which is lower than that of divide-and-conquer algorithms (8.48 %). As one more example, the average number of IPB in the benchmarks using backtracking algorithms is 9.5 whereas the number of IPB for the other benchmarks using divide-and-conquer, dynamic and branch-and-bound are respectively 6.64, 8.22 and 7.53 (shown in Fig. 12). Therefore, the presence of the pruning and the low rate of vulnerable instructions can improve the error-derating capability of backtracking algorithms. The results of the error-injection experiments shown in Fig. 4 illustrate that backtracking algorithms have 10.63 % more resiliency than the resiliency of divide-and-conquer algorithms.

iv. Branch-and-Bound Algorithms

Branch-and-bound algorithms are considered as an improved version of backtracking algorithms [13, 22, 45]. The bounding technique is used in branch-and-bound algorithms to avoid generating sub-trees which cannot produce an answer. Figure 18 illustrates a general structure of branch-and-bound algorithms. The *bound* function is used in an algorithm to decide if a node is promising or not.

Structural Features: The bounding technique is used to speed up branch-and-bound algorithms. As mentioned in Subsection 4.1, this technique is considered as a potential source of error-deration.

The bounding inherently and potentially results in the production of DUC and equivalent branches. The branch instruction mentioned in line 9 of Fig. 18 determines whether a node is considered as promising in the solution tree or not. The instructions following line 9 may be pruned and consequently become DUC.

Rate of Vulnerable Instructions: As shown in Fig. 13, the rate of function-calls in branch-and-bound algorithms is 1.58 % whereas the rates of the function-calls in divide-and-conquer and backtracking algorithms are respectively 8.48 % and 1.75 %. Moreover, as shown in Fig. 14, the average number of system-calls in the branch-and-bound algorithms is 63.96 K whereas the average numbers of system-calls in divide-and-conquer and backtracking algorithms are respectively 266.26 K and 81.92 K. Furthermore, the average number of IPB in the branch-and-bound algorithms is 7.53, which is higher than that of divide-and-conquer algorithms (6.64).

It should be noted that the presence of bounding technique and the low rate of vulnerable instructions in the branch-and-bound algorithms will result in an increase in the inherent resiliency. Regarding the results of the experiments, the implemented programs which rely on branch-and-bound algorithms have about 9.32 % higher resiliency than the resiliency of divide-and-conquer algorithms (shown in Fig. 4).

Figure 19 illustrates the average results of the error-injection experiments in the different benchmarks which used different algorithms- designing-techniques. Each data series in Fig. 19 illustrates the resiliency of different versions of each benchmark. On average, all the benchmarks using divide-and-conquer algorithms have approximately 28.87 %

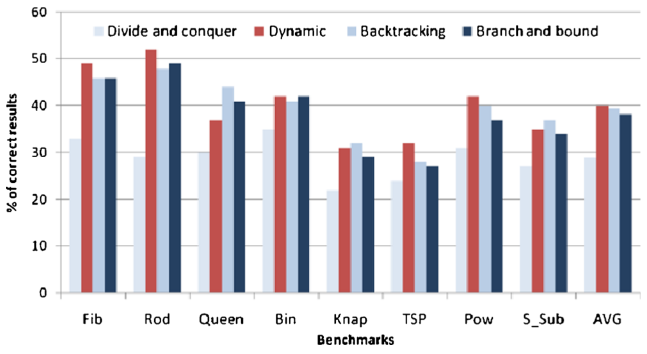


Fig. 19 The average results of error-injection experiments on different benchmarks

resiliency. The resiliencies of other benchmarks using dynamic, backtracking and branch-and-bound algorithms are respectively 40.00 %, 39.50 % and 38.12 %. The results of experiments indicate that the benchmarks using dynamic algorithms have about 12 % higher resiliency than those benchmarks using divide-and-conquer. Figure 20 demonstrates the effect of different algorithm-designing techniques on the rate of error-deration.

Regarding the results of experiments, shown in Figs. 19 and 20, it is possible to enhance the resiliency of an algorithm by means of potential sources without using an external redundancy. For example, the resiliency and execution time of the dynamic algorithm for the *TSP* benchmark are respectively 32 % and $O(n^2\ 2^{n-1})$ whereas the same parameters for the divide-and-conquer algorithm are respectively 24 % and $O((n-1)!)$. In other words, dynamic algorithm for the *TSP* benchmark enhances the performance and also resiliency. It should be maintained that software designers and programmers can develop a highly resilient program if they select a most fitting algorithm-designing technique without external redundancy.

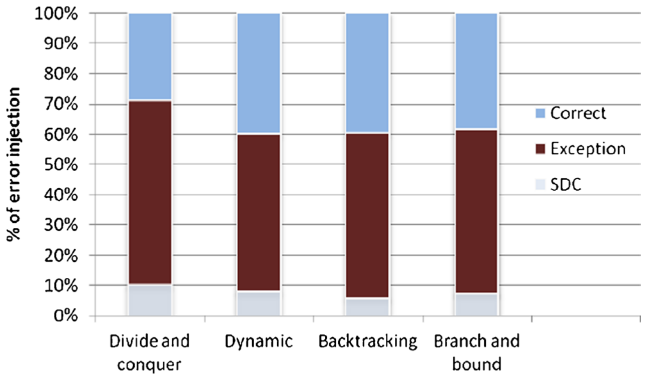


Fig. 20 The average effects of different algorithm-designing techniques on the rate of error-deration

As mentioned earlier in Subsection 4.1, the error-derating rate of an algorithm depends upon the following potential features:

1. The rates of dynamically-useless and quality-related codes and equivalent branches.
2. The rates of branch instructions, load/store instructions, function calls and system calls (as vulnerable instructions).

An inherent increase in the rate of the first feature and a decrease in the rate of the second feature will enhance the rate of error deration. A decrease in the rates of branch instructions, load/store and call instructions leads to a reduction of resource usage. The rates of vulnerable instructions in each benchmark when implemented by divide-and-conquer and dynamic algorithms were compared in Table 5. Furthermore, Table 6 gives the execution time of the benchmarks included in Table 5. In Table 5, v1 columns indicate the first versions of the benchmarks in which divide-and-conquer algorithm has been used. All the v2 columns stand for the second versions where dynamic algorithm has been used. As case in point, the rates of load/store and call instructions in the benchmarks implemented by dynamic algorithm are 2.90 % and 7.01 % respectively lower than the corresponding divide-and-conquer benchmarks.

The high rates of load/store instructions, system calls, function calls and branch instructions in divide-and-conquer benchmarks indicate higher resource usage for these benchmarks than dynamic benchmarks. Also, all the benchmarks implemented by dynamic algorithm, except for *n-Queen,* have lower execution time than the respective divide-and-conquer benchmarks. However, benchmarks implemented by dynamic algorithms have potentially higher resiliency than those implemented by divide-and-conquer algorithms. Low rates of branch, load/store and call instructions indicate that benchmarks based on dynamic algorithm have low resource usage though their resiliency is inherently high. To sum it up, it should be pointed out that using potential features in an algorithm to improve inherent resiliency has no significant effect on resource usage.

Effect of the Location of Error-Occurrence on the Deration Rate: In this study, we have investigated the error-derating capability of different algorithm-designing techniques. The rate of deration at algorithm level depends upon the rates of dynamically-

Table 5 The resiliency and rate of vulnerable instructions in the benchmarks implemented by divide-and-conquer and dynamic algorithms. The first and second versions of each benchmark were implemented by divide-and-conquer and dynamic algorithms

Benchmark	Branch instruction rate (%)		Call instruction rate (%)		Load/store instruction rate (%)		Resiliency (%)	
	V1	V2	V1	V2	V1	V2	V1	V2
Fib	23.4	13.2	7.3	2.01	32.2	21.4	33	49
Bin	12.6	35.7	13.7	1.2	53.5	51.7	35	42
Pow	32.1	31.4	2.1	1.3	36.9	36.7	31	42
TSP	44.4	43.3	16.2	4.8	43.7	44.8	24	32
Rod-c	32.2	11.3	22	0.4	31.1	26.1	29	52
Knap	53.8	11.6	2.1	0.8	37.8	27.3	22	31
Queen	13.9	9.6	1.9	0.09	20.3	33.8	30	37
S_sub	49.8	17.6	2.6	0.74	35.8	26.3	27	35
AVG.	32.775	21.7125	8.4875	1.4175	36.4125	33.5125	28.875	40

useless codes, quality-related codes and equivalent branches. An error which corrupts a dynamically-dead code or quality-related code in either *memory* or *microprocessor (ALU, FPU, PC, and LSU)* will more probably be derated at the execution time.

In order to examine the effect of the location of error-occurrence on the rate of deration, a new series of error-injection experiments targeting microprocessor were conducted. Indeed, the microprocessor was considered as the location of error-injection. The corresponding error-injection framework was developed in the *Simple-Scalar 4.0* toolset. Table 7 illustrates the locations of injections in the experiment. The injection locations include *PC, ALU and LSU*. A single-bit was flipped in the value of *PC*, output of *ALU* and the loaded data of *LSU* at a randomly selected clock-cycle during program execution.

In this experiment, a subset of sixteen benchmarks was selected from the benchmark set as described in Table 2. The selected subset includes the programs implemented by divide-and-conquer and dynamic algorithms. The main reason for the selection of divide-and-conquer and dynamic algorithms is that these algorithms have the lowest and highest derating capabilities respectively. Approximately 1,000 errors were injected at each location for each benchmark as illustrated in Table 7. Only one error at a time was injected in each program run. A single-bit was flipped in the randomly selected bit in each location. In total, 48,000 errors (1,000 errors × 3 injection location × 16 benchmarks) were injected in this experiment.

With regard to the results of the experiment, as shown in Fig. 21, when the errors are injected in *PC,* the benchmarks using dynamic algorithms have about 5.00 % higher resiliency than those benchmarks using divide-and-conquer algorithms; this figure is 8.25 % when the errors are injected in *ALU.* Table 8 illustrates the average deration rate in different programs when the errors were injected into the microprocessor. On average, the benchmarks using dynamic algorithm have 7.54 % higher resiliency than those benchmarks using divide-and-conquer when the errors were injected in the microprocessor structures (*PC, ALU and LSU*). The results of injecting errors both in memory and microprocessor are similar; this reveals that dynamic algorithms have higher resiliency than divide-and-conquer algorithms. To sum it up, dynamic based benchmarks have about 9.33 % higher resiliency than the divide-and-conquer based

Table 6 Execution times of all benchmarks implemented by divide-and-conquer and dynamic algorithms

	Fib	Bin	Pow	TSP	Rod-c	Knap	Queen	S_Sub
Divide-and-conquer	$O(2^n)$	$O(2^n)$	$O(n)$	$O((n-1)!)$	$O(2^n)$	$O(n2^n)$	$O(n!)$	$O(2^n)$
Dynamic	$O(n)$	$O(n^2)$	$O(lon)$	$O(n^2 2^{n-1})$	$O(n^2)$	$O(n*p)$	$O(8^n)$	$O(n)$

P: Capacity of knapsack

Table 7 The locations of error-injection in the microprocessor structure

Injection location	Description
Integer ALU	Output of ALU
LSU-d-out	Loaded data by LSU
PC	Program-counter register

benchmarks regardless of location of error occurrence (memory and microprocessor).

The researchers are fully aware of the fact that hardening memory blocks using ECC improves the rate of error-detection at memory level; as a result, this will improve the overall reliability of the system.

On the other hand, hardening the whole memory-blocks of a program using ECC imposes a considerable performance-overhead to the system. Furthermore, the microprocessor structures such as register and control logic are more susceptible to soft-errors than the memory blocks. Hence, using hardened memory blocks even for the entire program (code and data) could not protect it against errors affecting the registers and control logic.

In this research, we have investigated the inherent error-derating capability of different algorithms. Using an algorithm with high deration-rate will improve the program resiliency against errors which target either memory or microprocessor. Even if we assume that memory is hardened by ECC and errors affect the microprocessor, dynamic algorithms still have higher derating rate than the divide-and-conquer algorithms. As shown in Fig. 21, dynamic algorithms have higher error derating rate than the divide-and-conquer algorithms regardless of the location of error occurrence and the reliability of memory blocks. Indeed, the deration rate at algorithm level does not substantially depend on the reliability of the memory blocks; rather, it depends on the structural and algorithmic features.

Furthermore, the proposed method in this study can be used with different hardening techniques such as ECC. It is evident that using this method with ECC leads to further improvement in the system reliability.

4.3 Identification of Vulnerable Blocks

Invulnerable blocks, derating blocks, are the ones which inherently derate and mask soft-errors with high probability. In contrast, vulnerable blocks of a program probably propagate the impact of soft-errors to the program output. As a matter of fact, applying the redundancy technique only on the more vulnerable blocks imposes a lower performance overhead on the program. Hence, identifying the more vulnerable blocks of a program to soft-error is an important topic in the dependable-software development.

Soft-error deration is a semantic phenomenon which is related to the dynamic behavior of a program throughout its execution-time. Analyzing semantic dependencies among the instructions and blocks of a program can be regarded as an effective method for identifying more vulnerable blocks. Unlike the previous studies [3, 4, 12, 38, 44], the authors in the present study take the semantic and dynamic behavior of a program into account to identify error-derating and vulnerable blocks.

As mentioned in Subsection 4.1, memoization, pruning and bounding techniques potentially produce DUC, performance-related codes and equivalent branches in a program. Indeed, those blocks of a program that include DUC, performance related codes and equivalent branches are less vulnerable to errors and hence are considered as error-derating blocks. The dynamically useless blocks of a program can be located based on its underlying algorithm. Figure 22 shows the general structure of CFG for the programs using memoization, pruning or bounding techniques. The results of our experiments indicate that the probability of error deration in blocks 2 and 3 in Fig. 22 which include DUC and equivalent branches is more than 80 %. On the basis of this finding of the present study, it is argued that there should be a tradeoff between reliability and performance-overhead.

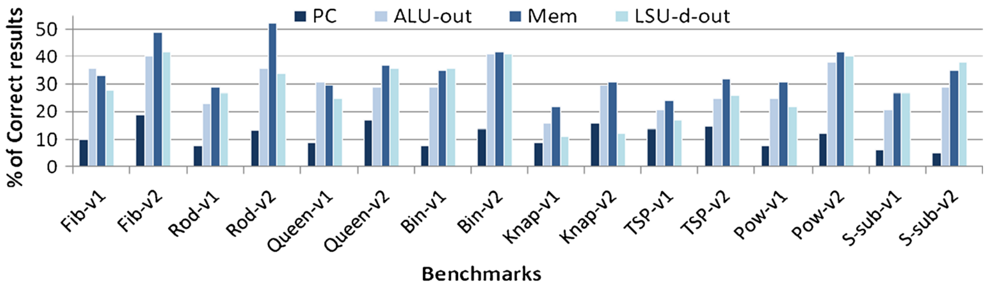


Fig. 21 The results of error-injection experiments on divide-and-conquer and dynamic based benchmarks

Table 8 The average results of error injection in the microprocessor structure

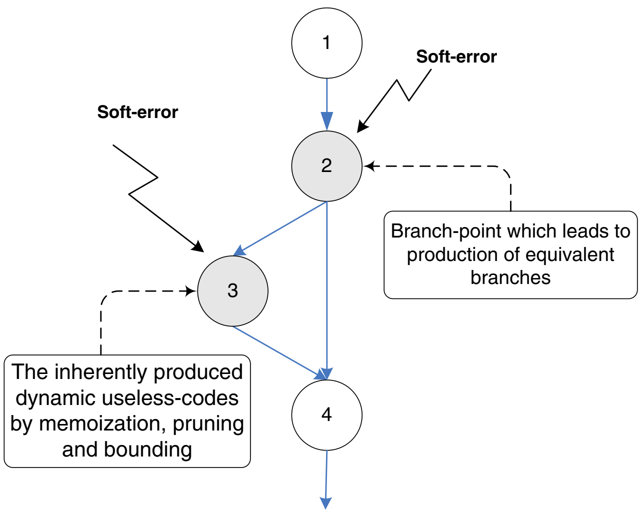
Program version	Rate of deration in microprocessor structures			
	PC	ALU-Out	LSU	AVG
Divide & conquer based programs	9.00 %	25.25 %	24.12 %	19.46 %
Dynamic based programs	13.87 %	33.50 %	33.62 %	27.00 %

In this subsection of the study, three series of error-injection experiments were conducted as follows:

- Errors were injected into those blocks which are identified as error-derating blocks by the underlying algorithm.
- Errors were injected into the randomly selected locations of the whole program.
- Errors were injected into those blocks which are not identified as error-derating blocks.

The first experiment acknowledges the derating capabilities of those blocks which contain DUC. The results of the second experiment demonstrated the overall derating capabilities of the programs. The results of the third experiment demonstrate the error-derating capabilities of vulnerable blocks in the programs.

- V2 refers to the benchmarks which used dynamic algorithms.
- V3 refers to the benchmarks which used backtracking algorithms.
- V4 refers to the benchmarks which used branch-and-bound algorithms.



1
Soft-error
Soft-error
2
Branch-point which leads to production of equivalent branches
3
The inherently produced dynamic useless-codes by memoization, pruning and bounding
4

Fig. 22 The production of dynamically useless code and equivalent branches by memoization, pruning and bounding in a program

Table 9 The average results of the error-injection experiments in different blocks of the different benchmarks

	Versions of benchmarks	Vulnerable blocks	Error-derating blocks	Whole program
Rate of failure (SDC+Exception)	V2	82 %	13 %	60 %
	V3	75 %	21 %	60 %
	V4	73 %	26 %	61 %

Table 9 demonstrates the results of error-injection experiments in the different blocks of different benchmarks. Table 7 illustrates the effect of injected errors on the more vulnerable blocks, error-derating blocks and on the entire blocks of the program. For example, the source codes of all programs using dynamic algorithms can be classified into two parts. The first part includes DUC and has close to 87 % inherent resiliency. The remainder of the source code has about 18 % inherent resiliency. This finding enables software designer to apply the redundancy techniques only on the more vulnerable blocks and hence develop highly reliable software with a minimal performance-overhead and complexities.

4.4 Statistical Analysis of the Results

We quantified the magnitude of error-derating capabilities of different algorithm-designing techniques in the previous subsections. We would like to examine whether there are statistically significant differences in the derating capabilities of different algorithm-designing techniques. ANOVA (analysis of variance) is used to check the significant differences among three or more independent groups [23, 31]; in this study, each group refers to an algorithm-designing technique. The level of confidence was set at 0.05 so as to minimize the role of chance for the likelihood of difference between the resiliencies of techniques. Table 10 illustrates the results of ANOVA for the groups of experimental data.

As Table 10 indicates, the *P* value (sig) for *between technique comparisons* is 0.009. That is, the obtained *F* ration of 4.68 illustrates a significant difference between at least two techniques. Therefore, the null hypothesis of no difference among groups is rejected indicating that at least two of the

Table 10 One way ANOVA across different algorithm-designing techniques

Resiliency	Sum of squares	df	Mean square	F	Sig.
Between groups	655.750	3	218.583	4.680	0.009
Within groups	1307.750	28	46.705		
Total	1963.500	31			

Table 11 LSD post-hoc comparison of results between algorithm-designing techniques as groups

Techniques (I)	Techniques (J)	Mean difference (I-J)	Sig.	95 % Confidence interval	
				Lower bound	Upper bound
1	2	−11.12500[a]	0.003	−18.1245	−4.1255
	3	−10.62500[a]	0.004	−17.6245	−3.6255
	4	−9.25000[a]	0.011	−16.2495	−2.2505
2	1	11.12500[a]	0.003	4.1255	18.1245
	3	0.50000	0.885	−6.4995	7.4995
	4	1.87500	0.588	−5.1245	8.8745
3	1	10.62500[a]	0.004	3.6255	17.6245
	2	−0.50000	0.885	−7.4995	6.4995
	4	1.37500	0.690	−5.6245	8.3745
4	1	9.25000[a]	0.011	2.2505	16.2495
	2	−1.87500	0.588	−8.8745	5.1245
	3	−1.37500	0.690	−8.3745	5.6245

The numbers 1, 2, 3 and 4 respectively refer to the divide-and-conquer, dynamic, backtracking and branch-and-bound techniques

[a] The mean difference is significant at the 0.05 level of confidence

techniques are significantly different from each other in terms of resiliency.

In order to determine the precise location of mean differences, the researcher applied a post hoc test. The LSD [31] was applied to make pair-wise comparisons among the different algorithm-designing techniques. The results of LSD post-doc testing are shown in Table 11. The greatest difference reached the magnitude of 11.125 between divide-and-conquer algorithm-designing technique and dynamic algorithm-designing technique. Post hoc comparisons confirmed that the divide-and-conquer and dynamic algorithms respectively have the lowest and the highest error-derating capabilities among all algorithms.

That is to say, the effectiveness of dynamic algorithms in terms of error-derating capability is significantly higher than the effectiveness of other algorithm-designing techniques. The SPSS [23] as a software package is used to statistical analysis of the results in the study.

5 Conclusion

This paper presented an experimental study for analyzing the error-derating capabilities of different algorithm-designing techniques. The findings of the study can be concluded as follows:

- We found that memoization, pruning and bounding are the potential sources of error-deration at algorithm level.
- The rates of branch, load/store and call instructions in four different algorithm-designing techniques were quantified and compared with each other. We found that dynamic and divide-and-conquer algorithms have the lowest and highest rates of vulnerable instructions respectively.
- The magnitude of the resiliency of four different algorithm-designing techniques were quantified and compared with each other. The resiliencies of divide-and-conquer, dynamic, backtracking and branch-and-bound techniques are respectively 28.8 %, 40.0 %, 39.5 % and 38.1 %. Knowledge of the resiliency of different algorithm-designing techniques can be helpful in dependable-system development. This finding can enable programmers and software designers to develop highly resilient programs without using external redundancy.
- We found that with regard to the underlying algorithmic features of a program, it is possible to identify the more vulnerable blocks. Indeed, we can classify the source code of a program into more vulnerable and less invulnerable parts. By applying the redundancy techniques only on the identified vulnerable blocks of a program, software designers and programmers will be able to obtain high reliability with a minimal performance-overhead.

As a follow up to the present study, interested researchers can analyze the following issues in future works:

- The effect of multiple-bit errors on the rate of error deration at algorithm level
- The effect of microprocessor characteristics on the rate of error deration at algorithm level
- the effect of different data-structures on the program resiliency
- the effect of different programming methods (like object-oriented, aspect-oriented and etc.) on the program resiliency
- the effect of different programming languages on the program resiliency

References

1. Ammann P, Mason G (2008) Introduction to software testing. Cambridge University Press, New York
2. Austin T, Larson E, Ernst D (2002) SimpleScalar: an infrastructure for computer system modeling. IEEE Comput 35(2):59–67
3. Benso A, Chiusano S, Prinetto P. Tagliaferri L (2000) C/C++ source-to-source compiler for dependable applications. In: IEEE International Conference on Dependable systems and Networks (DSN), June 2000
4. Benso A, Di Carlo S, Di Natale G, Prinetto P, Tagliaferri L (2003) Data criticality estimation in software application. In: International test conference, pp. 802–810, October 2003
5. Borodin D, Juurlink BHH (2010) Protective redundancy overhead reduction using instruction vulnerability factor. In: ACM international conference on computing frontiers, Italy, pp. 319–326, May 2010
6. Cook JJ, Zilles C (2008) A characterization of instruction-level error derating and its implications for error detection. In: IEEE international conference on dependable systems and networks (DSN), June 2008
7. Dixit A, Wood A (2011) The impact of new technology on soft error rates. In: Proceedings of the IEEE workshop on silicon errors in logic—system effects, Illinois University, March 2011
8. Engel H (1996) Data flow transformations to detect results which are corrupted by hardware faults. In: IEEE high-assurance system engineering workshop, pp. 279–285, October 1996
9. Fazeli M, Farivar R, Miremadi SG (2005) A software-based concurrent error detection technique for PowerPC processor-based embedded systems. In: 20th IEEE international symposium on defect and fault tolerance in VLSI Systems, pp. 266–274, October 2005
10. Hari SKS (2012) Low-cost program level detectors for reducing silent data corruptions. In IEEE international conference on Dependable Systems and Networks (DSN), June 2012
11. Henning JL (2006) SPEC CPU2006 benchmark descriptions. SIGARCH Comput Archit News 34(4):1–17
12. Hiller M, Jhumka A, Suri N (2001) An approach for analyzing the propagation of data errors in software. In: IEEE international conference on dependable systems and networks (DSN), July 2001
13. Horowitz E, Sahni S, Rajasekaran S (2008) Algorithms: design and analysis. Computer Science Press, ISBN: 0-929-30641-4
14. Karlsson J (1990) Reliability evaluation of a fault-tolerant computer for a multi-phased mission and a use of heavy-ion radiation for fault injection experiments, PhD Thesis, School of Electrical and Computer Engineering, Chalmers University of Technology
15. Karnik T, Hazucha P, Patel J (2004) Characterization of soft errors caused by single event upsets in CMOS process. IEEE Trans Dependable Secure Comput 1(2):128–143
16. Kleinberg J, Tardos E (2004) Algorithm design. Addison-Wesley, ISBN: 0-321-29535-8
17. Li X (2009) Exploiting inherent program redundancy for fault tolerance, PHD Thesis in University of Maryland
18. Lu JS, Li F, Degalahal V, Kandemir M, Vijaykrishnan N, Irwin MJ (2005) Compiler-directed instruction duplication for soft error detection. In: Design, automation and test in Europe conference, pp. 1056–1057, March 2005
19. Messer A (2004) Susceptibility of commodity systems and software to memory soft errors. IEEE Trans Comput 53(12):1557–1568
20. Miremadi G, Karlsson J, Gunneflo U, Torin J (1992) Two software techniques for online error detection. In: 22nd International symposium on fault-tolerant computing, pp. 328–335, July 1992
21. Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: 36th annual IEEE/ACM International symposium on micro architecture, pp. 29–40, December 2003
22. Neapolitan R, Naimipour K (2004) Foundations of algorithms using C++ pseudo code. Jones and Bartlett Publishers, ISBN: 0-763-72387-8
23. Norusis M (2008) SPSS 16.0 guide to data analysis. Prentice Hall, ISBN: 0-136-06136-2
24. Oh N, Mccluskey EJ (2002) Error detection by selective procedure call duplication for low energy consumption. IEEE Trans Reliab 51(4):392–402
25. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in super-scalar processors. IEEE Trans Reliab 51(1):63–75
26. Oh N, Shirvani PP, McCluskey EJ (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51(2):111–122
27. Pradhan DK (1996) Fault-tolerant computer system design. Prentice-Hall, ISBN:0-13-057887-8
28. Rebaudengo M, Sonza Reorda M, Torchiano M, Iolante M (2001) A source-to-source compiler for generating dependable software. In: IEEE international workshop on source code analysis and manipulation, pp. 33–42, November 2001
29. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (1999) Soft-error Detection through Software Fault-Tolerance Techniques. In: IEEE international symposium on defect and fault tolerance in VLSI systems, pp. 210–218
30. Reinhardt KS, Mukherjee S (2000) Transient fault detection via simultaneous multithreading. In: 27th annual international symposium on computer architecture, pp. 25–36, June 2000
31. Roberts MJ, Russo R (1999) A student's guide to analysis of variance. Routledge Publication, ISBN:0-415-16564-2
32. Rotenberg E (1999) Exploiting large ineffectual instruction sequences, Technical report, North Carolina State University, November 1999
33. Saggese GP, Wang NJ, Kalbarczyk ZT, Patel SJ, Iyer RK (2005) An experimental study of soft errors in microprocessors. IEEE Micro 25(6):30–39
34. Sahoo SK (2008) Using likely program invariants to detect hardware errors. In: IEEE International Conference on dependable systems and networks (DSN), June 2008
35. Savino A, Carlo SD, Politano G, Benso A, Dnatale G (2012) Statistical reliability estimation of microprocessor-based systems. IEEE Trans Comput 61(11):1521–1534
36. Sedgewick R (1998) Algorithms in C, Third edn. Addison-Wesley, ISBN 0-201-31452-5
37. Shivakumar P, Kistler M, Keckler S, Burger D, Alvisi L (2002) Modeling the effect of technology trends on soft error rate of combinational logic. In: International conference on Dependable Systems and Networks (DSN), June 2002
38. Shuguang F, Shantanu G, Ansari A, Mahlke S (2010) Shoestring: probabilistic soft-error resilience on the cheap. In: 15th international conference on architectural support for programming languages and operating systems, March 2010.
39. Slegel TJ, Averill RM, Check MA, Giamei BC, Krumm BW, Krygowski CA, Li WH, Liptay JS, MacDougall JD, McPherson TJ, Navarro JA, Schwarz EM, Shum K, Webb CF (1999) IBM's S/390 G5 microprocessor design. IEEE Micro 19(2):12–23
40. Sosnowski J (1994) Transient fault tolerance in digital systems. IEEE Micro 14(1):24–35
41. Sridharan V, Kaeli DR (2010) Using PVF traces to accelerate AVF modeling. In: Proceedings of the IEEE workshop on silicon errors in logic - system effects, Stanford, California, March 2010
42. Steininger A, Scherrer C (1997) On finding an optimal combination of error detection mechanisms based on results of fault injection experiments. In: 27th international symposium on fault-tolerant computing, USA, pp. 238–247, June 1997
43. Stephens C, Cogswell B, Gregory JH (1991) Instruction level profiling and evaluation of the IBM RS/6000. In: 18th international symposium on computer architecture, May 1991

44. Thaker D, Franklin D, Oliver J, Biswas S, Lockhart D, Metodi T, Chong FT (2006) Characterization of error-tolerant applications when protecting control data. In: IEEE international symposium on workload characterization, October 2006
45. Thomas H (2001) Introduction to algorithms. the MIT Press, ISBN: 0-262-03293-7
46. Wang F, Agrawal VD (2009) Soft error rates with inertial and logical masking. In 22nd international conference on VLSI design, January 2009
47. Wang N, Fertig M, Patel S (2003) Y-branches: when you come to a fork in the road, take it. In: International conference on parallel architectures and compilation techniques
48. Xiong L, Tan Q, Xu J (2011) Soft error mask analysis on program level. In: 10th international conference on network
49. Xu X, Li M (2012) Understanding soft error propagation using efficient vulnerability-driven fault injection. In IEEE international conference on dependable systems and networks (DSN), June
50. Yeh Y (1998) Design considerations in Boeing 777 fly-by-wire computers. In: 3rd IEEE International high-assurance systems engineering symposium, pp. 64–72, November 1998.
51. Zhang M, Shanbhag N (2004) A soft error rate analysis methodology. In IEEE/ACM International Conference on Computer-aided design, November 2004
52. Zhang B, Wang WS, Orshansky M (2006) FASER: fast analysis of soft error susceptibility for cell-based designs. In: 7th international symposium on quality electronic design, March 2006

Bahman Arasteh received his master's degree from Islamic Azad University of Arak, Iran, in 2006. He is currently working towards the Ph.D. degree in Islamic Azad University of Iran, Science and Research branch. His research interests include Software-Level Fault Tolerance, Software-Implemented Fault Injection, Reliability of Programming Languages, Compilers and Distributed Applications.

Seyed Ghassem Miremadi got his M.Sc. in Applied Physics and Electrical Engineering from Linköping Institute of Technology and his PhD in Computer Engineering from Chalmers University of Technology, Sweden, in 1984 and 1995, respectively. He is a professor of Computer Engineering at Sharif University of Technology. As fault-tolerant computing is his specialty, he initiated the "Dependable Systems Laboratory" at Sharif University in 1996 and has chaired the Laboratory since then. The research laboratory has participated in several research projects which have led to several scientific articles, conference papers and technical reports. Dr. Miremadi and his group have done research in Physical, Simulation-Based and Software-Implemented Fault Injection, Dependability Evaluation Using HDL Models, Fault-Tolerant Embedded Systems and Fault Tree Analysis. Dr. Miremadi was the Education Director (1997–1998) and the Head (1998–2002) of Computer Engineering Department at Sharif University and since 2002 is the Research Director of the department. He is a member of the IEEE Computer Society, IEEE Reliability Society and the Computer Society of Iran.

Amir Masoud Rahmani received his BS in Computer Engineering from Amir Kabir University, Tehran, in 1996, the MS in Computer Engineering from Sharif University of Technology, Tehran, in 1998 and the PhD degree in Computer Engineering from IAU University, Tehran, in 2005. Currently, he is an Associate Professor in the Department of Computer Engineering at the IAU University. He is the author/co-author of more than 120 publications in technical journals and conferences. His research interests are in the areas of distributed systems, ad hoc and wireless sensor networks and evolutionary computing.