Data Partitioning Techniques for Partially Protected Caches to Reduce Soft Error Induced Failures

Kyoungwoo Lee, Aviral Shrivastava, Nikil Dutt, and Nalini Venkatasubramanian

Abstract Exponentially increasing with technology scaling, soft errors have become a serious design concern in the deep sub-micron embedded systems. Partially Protected Cache (PPC) is a promising microarchitectural feature to mitigate failures due to soft errors in embedded processors. A processor with PPC maintains two caches, one protected and the other unprotected, both at the same level of memory hierarchy. The intuition behind PPC is that some data in the application is more prone to soft errors than others. By finding out the data more prone to soft errors and mapping only that to the protected cache, the failure rate can be significantly improved at minimal power and performance penalty. While the effectiveness of PPCs has been demonstrated on multimedia applications – where the multimedia data is inherently resilient to soft errors – no such obvious data partitioning exists for applications in general. This severely restricts the applicability of PPCs. This paper proposes profile-based data partitioning schemes that are applicable to applications in general and effectively reduce failures due to soft errors at minimal power and performance overheads. Our experimental results on HP iPAQ-like processor and memory configuration demonstrate that our algorithm efficiently reduces the failure rate by 47× on benchmarks from MiBench while incurring only 0.5% performance and 15% power overheads.

1 Introduction

System reliability is becoming the paramount concern in system design in the deep sub-micron era [1]. With technology scaling, i.e., smaller feature sizes, reduced voltage level, lower noise margins, etc., microprocessors are becoming increasingly prone to transient faults [10, 37]. A transient fault results in erroneous program states and eventually incorrect outputs, but it is temporary and non-destructive, i.e., resetting the device, restores normal behavior.

While transient faults may be caused due to several reasons, radiation-induced faults are responsible for more failures than all the other causes of transient faults combined [3]. Radiation-induced faults occur when a high energy radiation particle, e.g., an alpha particle, a neutron or a free proton, strikes the diffusion region of a CMOS transistor and produces charge, which results in toggling the logic value of the transistor. This phenomenon of change in the logic state of a transistor is called an *Upset*. An upset may result in a change in the architectural state of a processor. The changed architectural state of a processor is called an *Error*. An error can cause an observable difference in the behavior of the program, which is termed as a *Failure*.

Among all the microarchitectural features in a processor, on-chip caches are most susceptible to upsets. This is due to the fact that caches cover majority of chip area, and operate at much lower voltage than combinational circuits [8, 17]. In addition, while an upset in combinational circuits will become an error only if it is latched at the right moment, the absence of latching window masking in caches ensures that all upsets translate into errors. In fact, according to [19], more than 50% of errors occur in memories. Consequently, it is very important to prevent errors in memory structures.

Kyoungwoo Lee, Nikil Dutt, Nalini Venkatasubramanian
Department of Computer Science, School of Information and Computer Sciences, University of California, Irvine, CA 92697, USA, e-mail: {kyoungwl,dutt,nalini}@ics.uci.edu

Aviral Shrivastava
Department of Computer Science and Engineering, School of Computing and Informatics, Arizona State University, Tempe, AZ 85281, USA
e-mail: Aviral.Shrivastava@asu.edu

Several microarchitectural techniques have been proposed to reduce the impact of soft errors in memories, the most popular being the use of *Error Correction Codes* (ECC). While the ECC-based techniques are well suited for off-chip memories, they are not appropriate for caches, as they are highly sensitive to any power and performance overheads. In fact, implementing *Single-bit Error Correction and Double-bit Error Detection* (SECDED) codes in caches increases the cache access time by up to 95% [15] and power consumption by up to 22% [28]. Partially Protected Cache (PPC) was proposed by Lee et al. [14] to mitigate the impact of soft errors on caches. A PPC architecture has two caches, one *protected* against soft errors, and the other *unprotected*, at the same level of memory hierarchy. The intuition behind PPC is that when soft errors occur, some data is more likely to cause failures than others. By mapping only this data to the protected cache, the failure rate can be significantly reduced at minimal power and performance overheads. PPCs were demonstrated to be extremely effective (2 orders of magnitude reduction in failure rate at less than 7% performance and 10% energy penalty) for multimedia applications. In multimedia applications, the multimedia data itself is error-resilient. For example, in an image or video processing application, a soft error in the image or video only causes a slight loss in Quality of Service (QoS). In contrast, most other data, e.g, loop control variables, stack pointers, are not error-resilient. Any soft error in these variables may lead to a failure. However, no obvious data partitioning exists for applications in general. The absence of a data partitioning scheme for applications in general severely limits the applicability of PPC architectures.

In this paper, we propose schemes to partition the data of applications in general into the two caches of PPC architecture and achieve high reduction in failure rate, at minimal power and performance penalty. We develop and test several data partitioning algorithms. We find that Monte Carlo exploration is unable to find interesting data partitions. While Genetic Algorithm can efficiently search the exploration space, it does not achieve high reduction in failure rate. Our approach, DPExplore, can efficiently prune the search space, and uncover Pareto-optimal data partitions. Experimental results on the HP iPAQ h4600 [11]-like processor-memory subsystem running benchmarks from the MiBench suite [9] demonstrate that the PPC architectures can reduce the failure rate by $47\times$ with 0.5% performance and 15% energy penalty on average.

2 Background

The primary source of soft errors in digital CMOS circuits are cosmic radiation. Radiation-induced soft errors have been under investigation since late 1970s. Due to incessant technology scaling (decreasing supply voltage and shrinking feature size), the soft error rate (SER) has exponentially increased [10, 37], and now it has reached a point, where it has become a real threat to system reliability. Solutions to reduce the failures due to soft errors have been proposed at all levels of design hierarchy.

2.1 Packaging and Process Technology

Radioactive substances present in the packaging material are one of the sources of radiation that cause upsets. Therefore, techniques like purifying the packaging material, and hardening the semiconductors against radiation [4], have been proposed to reduce the occurrence of upsets. However, cosmic neutrons are the primary particles causing upsets, and high energy neutrons can pass through up to 5 feet of concrete [18]. Thus, packaging solutions are unable to shield from them completely.

Silicon-On-Insulator (SOI) [24, 31] has been proposed to reduce upsets by raising the *critical charge*, i.e., the minimal charge required for a device to keep data, by extending the depletion region or increasing the capacitance while maintaining or reducing the *collected charge*, i.e., the charge generated due to the radiation strikes, have been proposed. However, technology engineering requires the expense of additional process complexity, yield loss, and substrate cost [3].

2.2 Microarchitectural Solutions

While packaging and process technology solutions attempt to reduce the number of upsets, microarchitectural solutions attempt to reduce the number of upsets that translate into errors, and/or errors that result in failures. Solutions at the microarchitecture level can be categorized based on the components where they are applied: the combinational components, the sequential components, and the memory components.

Solutions for Combinational Logic Logic elements were considered more robust against soft errors than memory elements but many researchers predict that the logic SER will become one of main contributions to the system reliability [3, 33, 27]. The simplest and most effective way to reduce failures due to soft errors in combinational logic is Triple Modular Redundancy (TMR) [29], which uses three functionally equivalent replicas of a logic circuit and a majority 2-out-of-3 voter. But the overheads of hardware and power for conventional TMR exceed 200% [27]. Duplex modular redundancy [27, 20] is also possible but still it requires more than 100% area and power overheads without any optimization techniques. In order to reduce the high overheads in conventional redundancy techniques, Mohanram et al. in [20] presented the partial error masking by duplicating the most sensitive and critical nodes in a logic circuit based on the asymmetric susceptibility of the nodes to soft errors. Recently, Nieuwland et al. [27] proposed a structural approach analyzing the SER sensitivity of combinational logic to identify the SER critical components at circuits.

Solutions for Sequential Logic Temporal redundancy is another main approach that has been used to combat soft errors in circuits. In order to detect soft errors, [26] applied fine time-grain redundancy within the clock cycle greater than the duration of transient faults by using the temporal nature of soft errors. Krishnamohan et al. in [13] proposed the time redundancy methodology by using the timing slack available in the propagation path from the input to the output in CMOS circuits. A Razor flip-flop was presented in [7] to detect transient errors by sampling pipeline stage values with a fast clock and with a time-borrowing delayed clock.

Solutions for Memories By far, reducing soft errors in memories has been the most extensive research topic. Error detection and correction codes (EDC and ECC) have been widely investigated and implemented as the most effective schemes to detect and correct soft errors in memory systems. However, an ECC system consists of an encoding block as well as a decoding block responsible for detection and correction, and of extra bits storing parity values. Thus, ECC-based techniques consume extra energy and incur performance delay as well as additional area cost [28, 15, 29], and are therefore not suitable for caches. Thus, only a few processors such as the Intel Itanium processor [30] protect L2 and L3 caches with ECC, but we are not aware of any processor employing ECC based protection mechanism on L1-cache. This is mainly due to high overheads of ECC implementation [12, 21].

A cache scrubbing technique [22] has been proposed, that reads cache blocks periodically and fixes all single-bit errors, and which can avoid potential double-bit errors. Lowering supply voltage increases the probability of soft errors. To address this, [16] evaluated the drowsy cache and the decay cache exploiting voltage scaling and shut-down schemes, respectively, in order to decrease the power leakage. [16] also proposed an adaptive error correcting scheme to different cache data blocks, which can save energy consumption by protecting clean data less than dirty data blocks. [12] proposed the combined approach of parity and ECC codes to generate the reliable cache system in an area-efficient way. [25] presented an energy-efficient combined method with Hamming and Reed-Solomon codes in order to correct at least double-bit transient faults. However, they all exploit expensive error correcting codes in order to protect all the data unnecessarily. Recently, Cai et al. [6] profiled the effects of cache size selection on reliability and power consumption as well as performance by extensively simulating several benchmarks on different cache size configurations. They explored cache parameters to increase reliability while considering power and performance.

Partially Protected Cache Architecture PPC architecture was proposed in [14] and was demonstrated to be very effective in reducing the failure rate, while minimizing power and performance overheads. However, the effectiveness of PPCs has been demonstrated only on multimedia applications, and there is no known approach to use PPCs for general applications.

The contribution of this paper is in developing techniques to utilize PPC architectures for applications in general and establish PPC as an effective microarchitectural solution to mitigate failures due to soft errors.

3 Partially Protected Caches and Problem Definition

In a processor with *Partially Protected Cache* (PPC), the processor has two caches at the same level of memory hierarchy. As shown in Fig. 1, one of two caches is protected from soft errors, while the other one is unprotected. Any soft error protection mechanism can be implemented in the protected cache, e.g., increasing the thickness of oxide layer of the transistors in the cache, or adding redundancy logic like SECDED. To keep the access latencies of the protected cache and the unprotected cache the same, the protected cache is typically smaller than the unprotected cache.

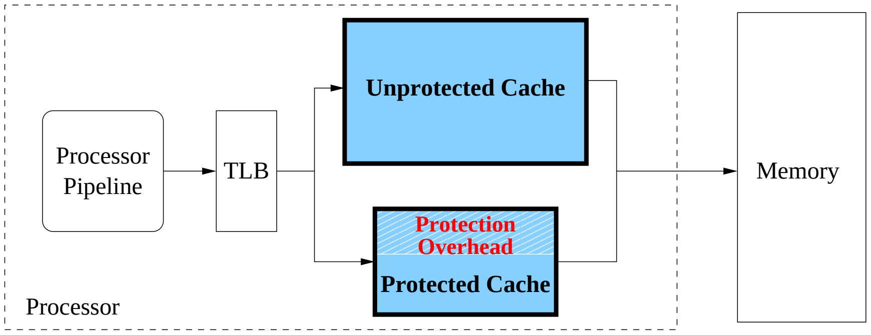


Fig. 1 Partially Protected Cache Architecture: one unproteced cache and the other protected cache at the same level of hierarchy

Each page in the memory is mapped exclusively to one of the caches in a PPC architecture. The page mapping is set as a page attribute by the compiler. The mapping of the pages present in the cache resides in the Translation Lookaside Buffer (TLB). On a cache access, first a TLB lookup is performed to find out if the page is present in the cache, and if so, in which one? Thus, only one cache lookup is performed per cache access.

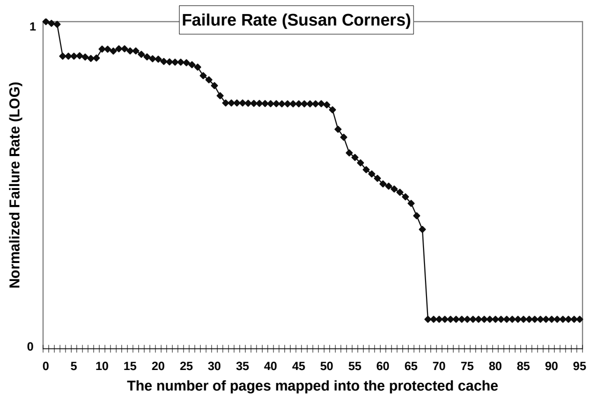


Fig. 2 Failure Rate Reduction by Mapping Pages into the Protected Cache

While PPC architectures can be very effective in reducing the failure rate with minimal performance and power overheads, the effectiveness hinges on the ability to partition the application data between the two caches. To motivate for the need and effectiveness of page partitioning to reduce the failure rate, we perform a small experiment. First we map all the application pages to the unprotected cache, and then move the pages to the protected cache one by one. Fig. 2 plots the failure rate at each step of this exploration for *susan corners*. The plot shows that the failure rate of the application drops rapidly as pages are moved from the unprotected cache to the protected cache. However, the pages have to be carefully moved to the small protected cache, as it is small; mapping too many pages to the small cache may increase the misses and result in a significant degradation of performance and increase in the energy consumption.

Therefore, the data partitioning problem is a multi-objective optimization problem in which we need to reduce the failure rate, at minimal performance degradation, and minimal increase in the energy consumption. Since, even medium sized applications use a large number of pages; our benchmarks [9] access 27 - 95, on average 56 pages. Owing to their exponential complexity, enumerative techniques (e.g. trying all the possible page partitions and picking up the best one) do not work.

We formulate our problem as: *Given an allowable performance degradation, determine the page partitioning that minimizes the failure rate of the application at minimal energy penalty.*

4 Our Approach

4.1 *Vulnerability: A Metric for Failure Rate*

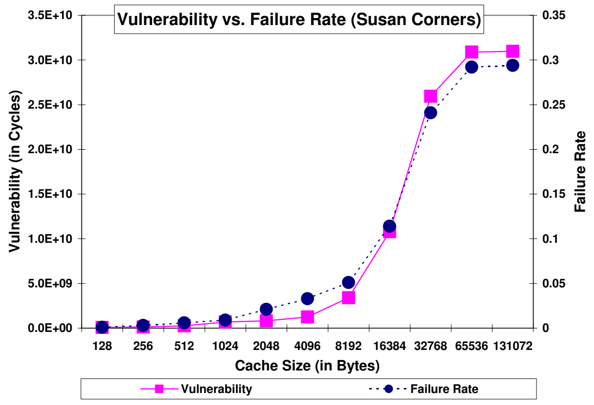


Fig. 3 Vulnerability and Failure Rate: vulnerability is a good metric for estimating failure rate

To choose pages to be mapped to the protected cache, we need a metric to quantitatively compare page partitions in terms of susceptibility to soft errors. We use the concept of vulnerability, proposed in [23, 2, 36, 38], to partition the data into the protected and unprotected caches in a PPC. We observe that if an error is injected in a variable that will not be used, the error does not matter. However, if the erroneous value will be used in the future, then it will result in a failure. Thus a data is defined to be **vulnerable** for the time it is in the cache until it is eventually read by the processor or written back to the memory. The vulnerability of an application is just the summation of the individual data vulnerability.

To validate our idea using vulnerability as a failure rate metric, we simulated the *susan corners* benchmark from MiBench suite on a modified *sim-outorder* simulator from SimpleScalar to model HP-iPAQ like system for various L1 cache sizes. Fig. 3 plots the *vulnerability* and the actual *failure rate* obtained by simulations. To estimate the actual failure rate, we injected soft errors for each execution of the benchmark, and calculated the number of failures out of a thousand executions. Each execution is defined as a success if it ends and returns the correct output. Otherwise, it is a failure. Fig. 3 shows that the shape of the *vulnerability* closely matches the failure rate curve. Other applications also show similar trends. On average, the error in predicting the failure rate using *vulnerability* metric is less than 5%. In this paper, we use *vulnerability* as the metric to estimate the failure rate, and perform automated design space exploration to decide the page partitioning between the two caches of a PPC. Reducing vulnerability can be contrary to performance improvement. For example, to reduce the vulnerability of data, data should not remain in the cache for long. It is better to evict and reload the reused data to reduce the vulnerability, but this may degrade performance. Therefore there is a fundamental trade-off between performance improvement and vulnerability reduction.

4.2 *Page Partitioning: DPExplore*

Fig. 4 outlines our DPExplore partitioning algorithm, which starts from the case when no page is mapped to the protected cache. In each step, pages are moved from the unprotected to the protected cache, to minimize the vulnerability under the runtime penalty. Our page partitioning algorithm takes two parameters: i. allowable runtime penalty (*rPenalty*), and ii. exploration width (*eWidth*), i.e., how many partitions are maintained as best configurations for the whole exploration. DPExplore uses *pCount*, the number of pages in a benchmark, and searches for page mappings that will suffer no more than the specified runtime penalty, while trying to minimize the vulnerability. DPExplore maintains a set of best page mappings found so far (Line 05) in $bestConfigs$, sorted in increasing order of vulnerabilities. After initialization, the algorithm goes into a forever loop in Line 07. It takes each existing best solution and tries to improve it by mapping a page to the protected cache (Lines 11-12). If the new page mapping is better than the worst solution in the $newBestConfigs$, then the new page

```
DPExplore(rPenalty, eWidth, pCount)
01: $pageMap0 = 0...0$
02: $rt, pow, vul = simulate(pageMap0)$
03: $config0 = (pageMap0, rt, pow, vul)$
04: for $(k = 0; k < eWidth; k++)$
05:     $bestConfigs.insert(config0)$
06: endFor
07: for (;;)
08:     $newBestConfigs = bestConfigs$
09:     for $(i = 0; i < eWidth; i++)$
10:         for $(j = 0; j < pCount; j++)$
11:             $testConfig = addPage(newBestConfigs[i], j)$
12:             $rt, pow, vul = simulate(testConfig.pageMap)$
13:             if $(rt < config0.rt \times \frac{100+rPenalty}{100})$
14:                 if $(vul < newBestConfigs[0].vul)$
16:                     $newBestConfigs.insert(testConfig, rt, pow, vul)$
17:                 endIf
18:             endIf
19:         endFor
20:     endFor
21:     for $(i = bestConfigs.length(); i > eWidth; i--)$
22:         $bestConfigs.delete[i-1]$
23:     endFor
24:     if $(newBestConfigs[0].vul < bestConfigs[0].vul)$
25:         $bestConfigs = newBestConfigs$
26:     else break;
27:     endIf
28: endFor
```

Fig. 4 DPExplore: an exploration algorithm for data partitioning

mapping is saved in the list. The loop in Lines 09-20 is one step of exploration. After each step, the new set of page mappings is trimmed down to exploration width (Lines 21-23). The termination criterion of the exploration is when an exploration step cannot find any better page mapping. In other words, no page can be mapped to the protected cache to improve vulnerability (Lines 24, 26) under the runtime penalty. Otherwise, the global collection of the best page mappings are updated.

Note that our exploration technique is a profile-based approach, which works well if the page mapping of application codes and input data does not change. Our proposal, DPExplore, is very effective for such applications.

5 Experiments

5.1 Setup

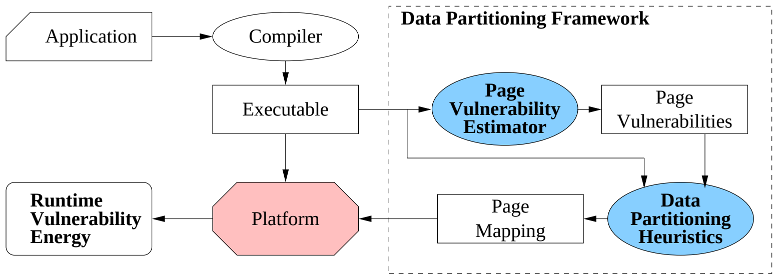

Application
Compiler
Executable
Data Partitioning Framework
Page Vulnerability Estimator
Page Vulnerabilities
Data Partitioning Heuristics
Page Mapping
Platform
Runtime Vulnerability Energy

Fig. 5 DPExplore Page Partitioning Framework for PPC Architectures

In order to demonstrate the effectiveness of DPExplore in exploring and discovering the partition with minimal vulnerability at minimal power and runtime[1] penalty, we have built an extensive simulation framework. The application is first compiled to generate an executable. The application is then profiled, and the *Page Vulnerability Estimator* calculates the vulnerability of each page accessed by the application. The pages are then sorted according to their vulnerabilities, and then *Data Partitioning Heuristics* partitions and maps the pages to the two caches in the PPC architecture. Through the simulations, *Data Partitioning Heuristics* finds out the page mapping with minimal vulnerability under the runtime con-

[1] Here runtime and performance are used interchangeably and represent the number of cycles for execution of an application

straint. Finally, the executable and the page mapping are provided to the platform, which runs the application and generates outputs such as runtime, energy consumption, and vulnerability.

The platform is modeled using *sim-outorder* simulator from the SimpleScalar toolchain [5]. The simulation parameters have been setup so as to model an HP iPAQ h4600 [11] like processor memory system. We model a PPC architecture consisting of a 4 KB of unprotected cache and a 256 bytes of protected cache with line size of 32 bytes, 4 way set-associativity, and FIFO cache replacement policy. This model protects one small cache with ECC-based technique such as Hamming Code. The overheads of power and delay for ECC protected caches are estimated and synthesized using the CACTI [32] and the Synopsys Design Compiler [35] as in [14]. And also SimpleScalar *sim-outorder* simulator has been modified to include the vulnerability computation. The memory subsystem includes the caches, external buses, and 2 off-chip SDRAMs. To estimate the memory subsystem energy consumption, we use the power models presented in [34].

The HP iPAQ is a wireless handheld device, and MiBench is the set of benchmarks that are representative of applications that run on wireless handheld devices [9]. MiBench suite is therefore the right set of benchmarks that are supposed to run on the iPAQ, and we choose them. However, we pick only those benchmarks in which the runtime difference between the case when all data is mapped to the 4 KB unprotected cache, and the case when all data is mapped to the 256 bytes protected cache in the PPC is more than 5%. This is to avoid benchmarks for which only the small protected cache is enough. Note that although some of the benchmarks in MiBench are multimedia applications (for which an obvious data partitioning exists), we use DPExplore to partition the data of **all** applications in the selected benchmark suite.

We compare the effectiveness of our approach DPExplore with two traditional exploration techniques,

Monte Carlo (MC) In MC, several page partitions are randomly generated and tested by simulation for their effectiveness in improving power, runtime and vulnerability.

Genetic Algorithm (GA) For GA, initially, we form a randomly generated sequence, representing a page mapping. At each successive generation, the superior sequences in terms of vulnerability are selected as the evolutionary page mappings through the simulation, where vulnerability, power, and runtime are evaluated. In order to generate the next sequence, we implemented two GA operations such as mutation and crossover. For the mutation operation, a pseudo-random number tells whether each page mapping in a sequence is modified or not. For the crossover operation, one point is selected in the current sequence and the bits are swapped on page mappings to generate the next sequence.

5.2 Results

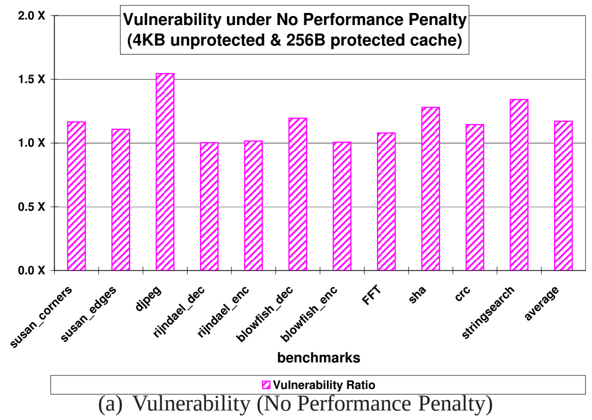
5.2.1 Effectiveness of DPExplore

We perform two kinds of experiments to demonstrate the effectiveness of DPExplore. In the first set of experiments, we find the page partition with the least vulnerability without any performance loss. Fig. 6(a) and Fig. 6(b) plot the vulnerability ratio and the memory subsystem energy ratio, respectively, of the least vulnerability page partition obtained by DPExplore. *Vulnerability Ratio* indicates the ratio of the vulnerability of the *base case* to the vulnerability discovered by DPExplore. Similarly, *Runtime Ratio* and *Energy Ratio* of the least vulnerability page partition obtained by DPExplore are presented in Fig. 6(b). Thus, each ratio greater than 1 implies the reduction of each metric. In case of no performance penalty, our heuristic algorithm can discover partitions with on average more than 1.2 times reduction in vulnerability, i.e., 1.2 in vulnerability ratio, and only about 3% energy overhead, i.e., 0.97 in energy ratio, over all benchmarks.

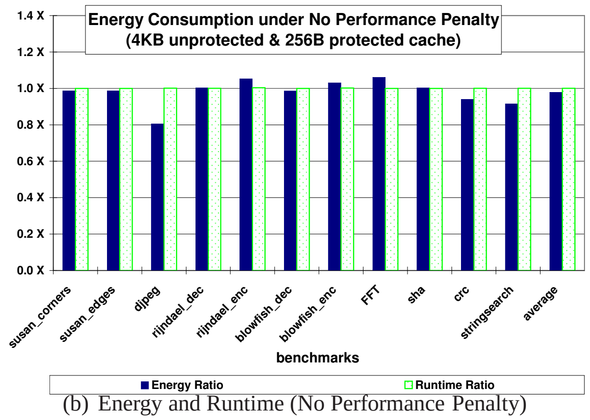
In the second experiment, we allow 5% performance degradation. Fig. 6(c) plots the vulnerability reduction and Fig. 6(d) plots the increase in energy consumption of the memory subsystem and the increase in runtime of the least vulnerability page partition obtained by DPExplore. We observe 47$\times$ reduction in vulnerability on average, along with only 0.5% degradation in runtime, and 15% increase in the total energy consumption of the memory subsystem. Compared to the case when all data are mapped to the protected 4 KB cache, i.e., the completely protected cache, the runtime and the energy consumption of the page partition with DPExplore are improved by 36% and 9%, respectively. Thus, even very small runtime degradation allows DPExplore to find page mappings that can significantly reduce the vulnerability.

5.2.2 Sensitivity of Vulnerability Reduction

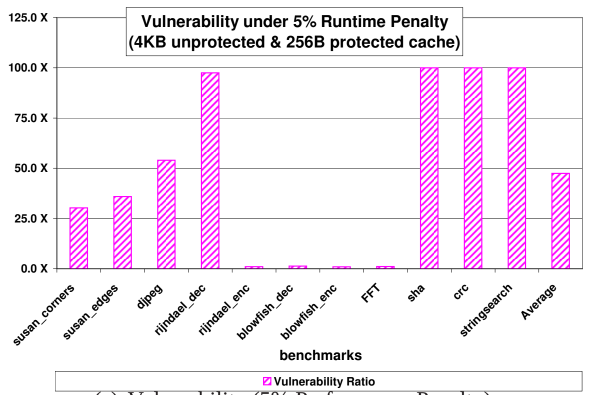
Next we study the effectiveness of vulnerability reduction with DPExplore by varying the allowable runtime penalty and the exploration width. Fig. 7(a) shows that as we increase the exploration width from 1 to 10, the average vulnerability



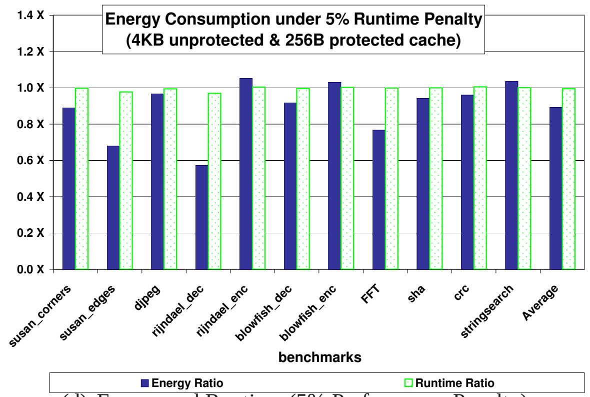
(a) Vulnerability (No Performance Penalty)



(b) Energy and Runtime (No Performance Penalty)

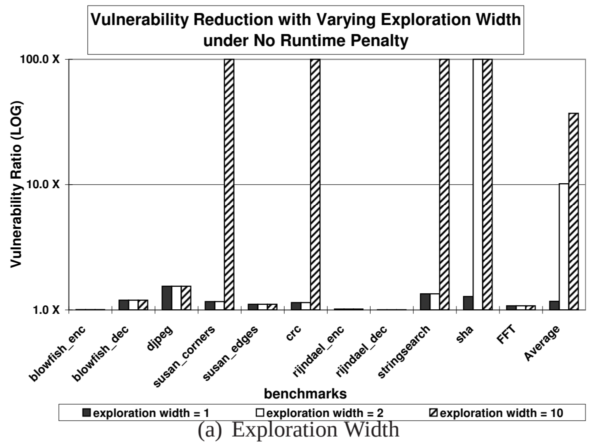


(c) Vulnerability (5% Performance Penalty)

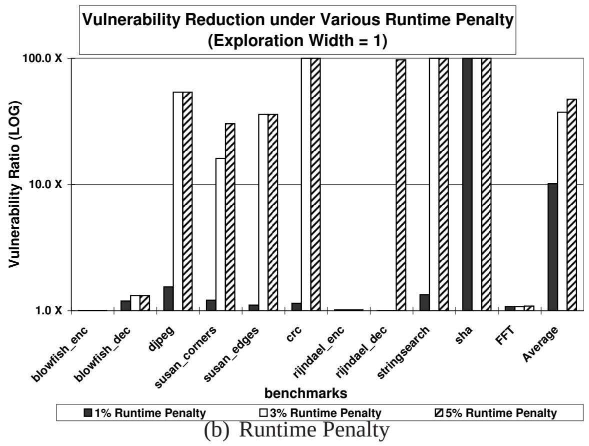


(d) Energy and Runtime (5% Performance Penalty)

Fig. 6 Evaluation under No Performance Penalty and 5% Performance Penalty: DPExplore can significantly reduce the vulnerability at minimal runtime and power



(a) Exploration Width

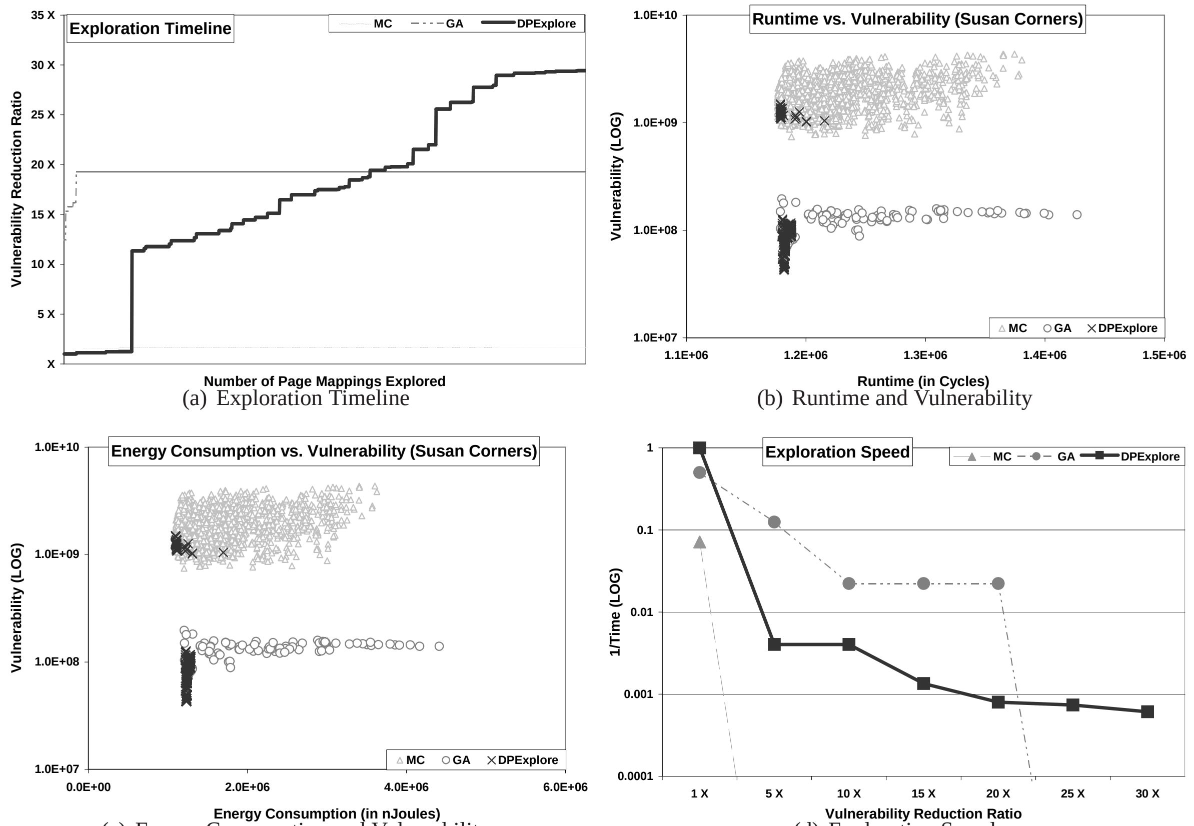


(b) Runtime Penalty

Fig. 7 DPExplore under Effective Parameters: Exploration Width and Runtime Penalty

reduction increases since larger width results in more page partitions to be explored. Fig. 7(b) shows that as we increase the allowable runtime penalty, from 1% to 5%, the average vulnerability reduction increases since relaxing the runtime constraint also increases the exploration space. Interestingly *sha* has a page partitioning, which achieves high vulnerability reduction even without runtime penalty. This is because the data reuse is inherently lower in *sha*, therefore mapping most application data to the small protected cache does not degrade the runtime too much, but reduces the vulnerability significantly.

5.2.3 Comparison with Other Explorations



(a) Exploration Timeline (b) Runtime and Vulnerability

(c) Energy Consumption and Vulnerability (d) Exploration Speed

Fig. 8 Exploration by MC, GA and DPExplore: DPExplore can effectively explore the design space

We detail the results of exploration using MC, GA, and DPExplore over the *susan corners* benchmark, when DPExplore is configured for 5% runtime penalty, and exploration width 2. Fig. 8(a) plots the vulnerability as the exploration progresses for MC, GA, and DPExplore. The plot shows that while MC is ineffective, GA improves vulnerability by about 20×, but DPExplore consistently finds better page mappings and is eventually able to reduce vulnerability by about 30×.

Fig. 8(b) and Fig. 8(c) plot the runtime, energy consumption, and vulnerability of the page partitions searched by MC, GA, and DPExplore. Note that the y-axis in these graphs – the vulnerability scale – is logarithmic. The most important observation that we make from these graphs is that DPExplore searches much more useful page mappings (low vulnerability with low runtime and energy overheads), as compared to MC and GA. We allow each exploration technique to evaluate 1,900 page mappings. Thus, in total there are 5,700 page mappings. Out of them only 83 are Pareto-optimal. A page mapping is Pareto-optimal, if it is no worse than any other configuration in all the three dimensions, i.e., runtime, vulnerability and energy. Out of these 83 Pareto-optimal page mappings, 68 were first drawn from DPExplore searches (82%), 12 came from GA (14%), and only 3 were discovered by MC (4%). This Pareto-optimal observation demonstrates the effectiveness of our algorithm as compared to MC and GA. The main reason for the effectiveness of DPExplore as compared to MC and GA explorations is that MC and GA ignore the effects of partitioning on the runtime and energy consumption.

Finally, we compare the speed of the various exploration algorithms. Fig. 8(d) plots the speed of exploration, i.e., inverse of the number of page partitions explored to achieve a required vulnerability reduction. The plot shows that MC is quite ineffective. Among GA and DPExplore, GA is a faster approach when low reduction in vulnerability is required, but it is unable to achieve high reductions in vulnerability. This is where, our approach is really effective.

6 Summary

Owing to the incessant technology scaling, soft errors, especially in caches are becoming a critical design concern for system reliability. Partially Protected Cache (PPC) architecture has been proposed as an effective architectural means of improving system reliability without much power and performance penalty. PPC architectures maintain two caches, one protected and the other unprotected at the same level of hierarchy. However, the challenge in exploiting PPC architectures is in mapping pages among the two caches. While page mapping schemes have been proposed for multimedia applications, there is no page mapping scheme for general applications. The page mapping space is huge, and existing random techniques are unable to identify and explore the page mappings that lead to low vulnerability. In this paper, we develop DPExplore, a page mapping algorithm that can effectively and efficiently explore to find page mappings that result in 47 times reduction in vulnerability, i.e., in failure rate, at only 0.5% performance and 15% energy penalty on average. The main contribution of DPExplore is, that it increases the applicability of PPC architectures, and establishes PPC as the solution of choice to improve failure rates of cache-based architectures.

References

[1] International technology roadmap for semiconductors 2005.
[2] G.-H. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli. Balancing performance and reliability in the memory hierarchy. In *ISPASS*, 2005.
[3] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, pages 258–266, 2005.
[4] M. P. Baze, S. P. Buchner, and D. McMorrow. A digital cmos design technique for seu hardening. *IEEE Trans. on Nuclear Science*, 47(6):2603–2608, Dec 2000.
[5] D. Burger and T. M. Austin. The SimpleScalar Tool Set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.
[6] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy. Cache size selection for performance, energy and reliability of time-constrained systems. In *ASP-DAC*, 2006.
[7] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO-36*, pages 7–13, 2003.
[8] J. Gaisler. Evaluation of a 32-bit microprocessor with builtin concurrent error-detection. In *FTCS*, 1997.
[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Fourth IEEE Workshop Workload Characterization*, Dec 2001.
[10] P. Hazucha and C. Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. on Nuclear Science*, 47(6):2586–2594, 2000.
[11] Hewlett Packard, http://www.hp.com. *HP iPAQ h4000 Series - System Specifications*.
[12] S. Kim. Area-efficient error protection for caches. In *DATE'06*, Mar 2006.
[13] S. Krishnamohan and N. R. Mahapatra. An efficient error-masking technique for improving the soft-error robustness of static cmos circuits. In *SOCC*, Sep 2004.
[14] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *CASES*, Oct 2006.
[15] J.-F. Li and Y.-J. Huang. An error detection and correction scheme for rams with partial-write function. In *MTDT'05*, pages 115–120, 2005.
[16] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: A data cache perspective. In *ISLPED*, Aug 2004.
[17] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *FTCS-24*, 1994.
[18] R. Mastipuram and E. C. Wee. *Soft Errors' Impact on System Reliability*. http://www.edn.com/article/CA454636, Sep 2004.
[19] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, Feb 2005.
[20] K. Mohanram and N. A. Touba. Partial error masking to reduce soft error failure rate in logic circuits. In *DFT03*, pages 433–440, 2003.
[21] K. Mohr and L. Clark. Delay and area efficient first-level cache soft error detection and correction. In *ICCD*, 2006.
[22] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *PRDC'04*, 2004.
[23] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, Dec 2003.
[24] O. Musseau. Single-event effects in soi technologies and devices. *IEEE Trans. on Nuclear Science*, Apr 1996.
[25] G. Neuberger, F. D. Lima, L. Carro, and R. Reis. A multiple bit upset tolerant sram memory. *ACM Trans. on Design Automation of Electronic Systems*, 8(4), Oct 2003.
[26] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VTS'99*, 1999.
[27] A. K. Nieuwland, S. Jasarevic, and G. Jerin. Combinational logic soft error analysis and protection. In *IOLTS06*, 2006.
[28] R. Phelan. Addressing soft errors in arm core-based designs. Technical report, ARM, 2003.
[29] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996. ISBN 0-1305-7887-8.
[30] N. Quach. High availability and reliability in the itanium processor. *IEEE MICRO*, pages 61–69, Sep–Oct 2000.
[31] P. Roche, G. Gasiot, K. Forbes, Oapos, V. Sullivan, and V. Ferlet. Comparisons of soft error rate for srams in commercial soi and bulk below the 130-nm technology node. *IEEE Trans. on Nuclear Science*, 50(6), Dec 2003.
[32] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. In *WRL Technical Report 2001/2*, 2001.
[33] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *DSN02*, 2002.
[34] A. Shrivastava, I. Issenin, and N. Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *CASES'05*, 2005.
[35] Synopsys Inc., Mountain View, CA, USA. *Design Compiler Reference Manual*, 2001.
[36] S. Wang, J. Hu, and S. G. Ziavras. On the characterization of data cache vulnerability in high-performance embedded microprocessors. In *SAMOS*, 2006.
[37] F. Wrobel, J. M. Palau, M. C. Calvet, O. Bersillon, and H. Duarte. Simulation of nucleon-induced nuclear reactions in a simplified sram structure: Scaling effects on seu and mbu cross sections. *IEEE Trans. on Nuclear Science*, 48(6):1946–1952, 2001.
[38] W. Zhang. Computing cache vulnerability to transient errors and its implication. In *DFT'05*, 2005.