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. 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 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 demonstrate that our algorithm reduces the failure rate by 47× on benchmarks from MiBench while incurring only 0.5% performance and 15% power overheads.

1 Introduction

Reliability is becoming the paramount concern in system design in the deep sub-micron era [1]. With technology scaling, i.e., smaller feature size, lower voltage level, etc., microprocessors are becoming increasingly prone to transient faults [9]. A transient fault results in erroneous program states and eventually incorrect out-

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

puts, but it is non-destructive, i.e., resetting the device, restores normal behavior. While transient faults occur due to several reasons, radiation is more responsible for transient faults than all the other causes combined [4]. 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 [7, 16]. In addition, while an upset in combinational circuits becomes 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. Indeed, more than 50% of errors occur in memories [17]. Consequently, it is very important to prevent errors in memory structures.

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 inappropriate for caches, as they are highly sensitive to any power and performance overheads. In fact, implementing an ECC scheme in caches increases the cache access time by up to 95% [14] and power consumption by up to 22% [24]. Partially Protected Cache (PPC) was proposed by Lee et al. [13] 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 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, etc., are not error-resilient. Any soft error in these variables may lead to a failure. However, no obvious data partitioning exists for general applications. 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 general applications into the two caches of PPC architecture and to achieve high reduction in failure rate, at minimal power and performance penalty. We develop and test several data partitioning algorithms. Monte Carlo exploration is unable to find interesting data partitions. While Genetic Algorithm efficiently searches the exploration space, it does not achieve high reduction in failure rate. Our approach, *DPExplore*, efficiently prunes the search space, and uncovers Pareto-optimal data partitions. Experimental results on the HP iPAQ h4600 [10]-like processor-memory subsystem running

benchmarks from MiBench [8] demonstrate that the PPC architectures reduce the failure rate by 47× with 0.5% performance and 15% energy penalty on average.

2 Related Work

Radiation-induced soft errors have been under investigation since late 1970s. Due to incessant technology scaling, soft error rate (SER) has exponentially increased [9], and now it has reached a point, where it becomes a real threat to system reliability. 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 soft errors will become one of main contributions to the system unreliability [4, 23, 28]. The simplest and most effective way to reduce failures due to soft errors in combinational logic is Triple Modular Redundancy (TMR) [25], which typically uses three functionally equivalent replicas of a logic circuit and a majority voter. But the overheads of hardware and power for conventional TMR exceed 200% [23]. Duplex redundancy [18, 23] is also available but 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 [18] presented a partial error masking by duplicating the most sensitive and critical nodes in a logic circuit based on the asymmetric susceptibility of nodes to soft errors. Recently, Nieuwland et al. in [23] proposed a structural approach analyzing the SER sensitivity of combinational logic to identify the 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, Nicolaidis in [22] applied fine time-grain redundancy within the clock cycle greater than the duration of transient faults by using the temporal nature of soft errors. Similarly, Anghel et al. in [2] exploited the temporal nature to detect timing errors and soft errors by means of time redundancy. Krishnamohan et al. in [12] 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 [6] 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 [14, 24, 25], and are therefore not suitable for caches. Thus, only a few processors such as the

Intel Itanium processor [26] 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 [11, 19].

Mukherjee et al. in [20] proposed a cache scrubbing technique that can avoid potential double-bit errors by reading cache blocks periodically and fixing all single-bit errors. Li et al. in [15] evaluated the drowsy cache and the decay cache exploiting voltage scaling and shut-down schemes, respectively, in order to efficiently decrease the power leakage. They 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. Kim in [11] proposed the combined approach of parity and ECC codes to generate the reliable cache system in an area-efficient way. However, they all exploit expensive error correcting codes in order to protect all the data unnecessarily. Recently, Sugihara et al. in [30] presented a task scheduling method to dynamically switch the operation modes between the performance and vulnerability in cache architectures of multiprocessor systems.

Partially Protected Cache Architecture Lee et al. in [13] proposed PPC architecture and demonstrated the effectiveness in reducing the failure rate with minimal power and performance overheads. However, the effectiveness of PPCs has been limited 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

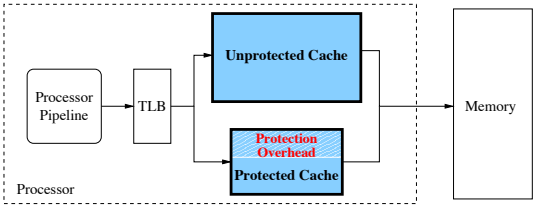


Fig. 1 Partially Protected Cache Architecture: one protected cache and the other unprotected cache at the same level of hierarchy

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 is unprotected. Any protection mechanism can be implemented in the protected cache, e.g., increasing the thickness of oxide layer of the transistors, or adding redundancy logic like a Hamming Code [25]. The protected cache is typically smaller than the unprotected cache to keep the access latencies of both caches the same. 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.

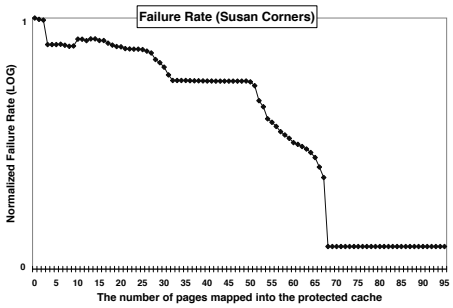


Fig. 2 Failure Rate Reduction by Moving Pages from the Unprotected Cache into the Protected Cache One by One in a PPC

While PPC architectures are 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 in a PPC. To motivate for the need 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*, and shows that the failure rate drops rapidly as pages are moved from the unprotected cache to the protected cache. However, the pages have to be carefully moved to the protected cache, as it is small; mapping too many pages to the small cache increases the misses and results in significant penalties of performance and energy consumption due to frequent memory accesses. Therefore, the data partitioning is a multi-objective optimization problem in which we need to reduce the failure rate, at minimal overheads of performance and energy consumption. Since, even medium sized applications use a large number of pages; our benchmarks from [8] access 27 - 95 pages. Owing to its exponential complexity, enumerative techniques (e.g. trying all the possible page partitions) do not work.

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

4 Our Approach

4.1 Vulnerability: A Metric for Failure Rate

To partition pages for a PPC architecture, we need a metric to quantitatively compare page partitions in terms of susceptibility to soft errors. We use the concept of

vulnerability, proposed in [3, 21], to partition the data into the protected and unprotected caches in a PPC. If an error is injected in a variable that will not be used, the error does not matter. However, if the erroneous value is 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 unprotected cache until it is eventually read by the processor or written back to the memory. The vulnerability of an application is the summation of the individual data vulnerability measured in cycles to present the vulnerable time of this data.

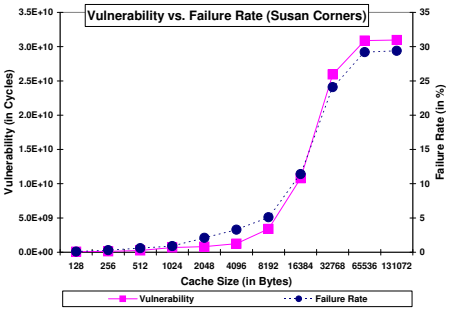


Fig. 3 Vulnerability and Failure Rate: vulnerability is a good metric for estimating failure rate

To validate our idea using vulnerability as a failure rate metric, we simulated the *susan corners* benchmark from [8] on a modified *sim-outorder* simulator from SimpleScalar [5] to model HP-iPAQ [10] like system for various L1 cache sizes. Our modified simulator calculated the vulnerability for each cache size as discussed above. To estimate the failure rate, we injected soft errors on data caches for each run of the benchmark, counted the number of failed runs out of a thousand runs, and calculated the failure rate in %. Each run is defined as a success if it ends and returns the correct output. Otherwise, it is a failure. Fig. 3 plots the *vulnerability* and the *failure rate* obtained by simulations and 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 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 degrades 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 three inputs: (i) allowable runtime penalty (*rPenalty*), (ii) exploration width (*eWidth*), the number of partitions maintained as best configurations for the whole exploration, and (iii) *pCount*, the number of pages in a benchmark. DPExplore searches for page mappings that will suffer no more than the specified *rPenalty*, while trying to minimize the vulnerability. DPExplore maintains a set of best page mappings found so far (Line 05) in $bestConfigs$, sorted in an increasing order of vulnerability. 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 mapping is saved in the list. The loop in Lines 09-21 is one step of exploration. After each step, the new set of page mappings is trimmed down to exploration width (Lines 22-24). 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 25, 27) under the runtime penalty. Otherwise, the global collection of the best page mappings are updated (Line 26).

```
DPExplore(rPenalty, eWidth, pCount)
01: pageMap0 = 0...0
02: runtime, power, vulnerability = simulate(pageMap0)
03: config0 = (pageMap0, runtime, power, vulnerability)
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:             runtime, power, vulnerability = simulate(testConfig.pageMap)
13:             if (runtime < config0.runtime × (100+rPenalty)/100)
14:                 if (vulnerability < newBestConfigs[0].vulnerability)
16:                     newBestConfigs.insert(testConfig, runtime, power, vulnerability)
17:                     newBestConfigs.sort()
18:                 endIf
19:             endIf
20:         endFor
21:     endFor
22:     for (i = newBestConfigs.length(); i > eWidth; i--)
23:         newBestConfigs.delete[i − 1]
24:     endFor
25:     if (newBestConfigs[0].vulnerability < bestConfigs[0].vulnerability)
26:         bestConfigs = newBestConfigs
27:     else break;
28:     endIf
29: endFor
```

Fig. 4 DPExplore: an exploration algorithm for data partitioning

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

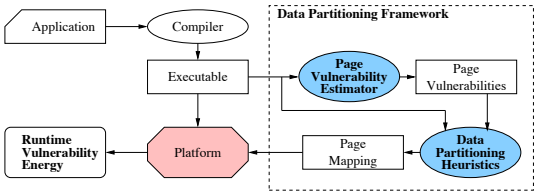


Fig. 5 DPExplore Page Partitioning Framework for PPC Architectures

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 constraint. 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 [10] 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 an ECC-based technique such as a Hamming Code [25]. The overheads of power and delay for ECC protected caches are estimated and synthesized using the CACTI [27] and the Synopsys Design Compiler [31] as in [13]. 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 [29].

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 [8]. 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 cases when all data is mapped to the

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

4 KB unprotected cache, and when all data 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.

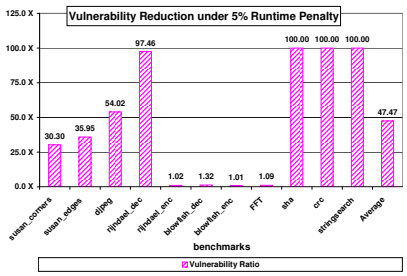
We compare the effectiveness of our approach DPExplore with two traditional exploration techniques,

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

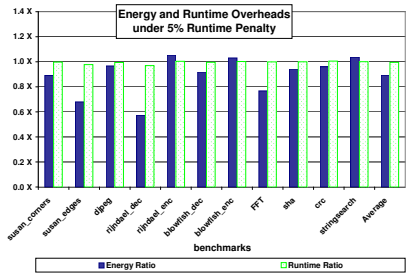
Genetic Algorithm (GA) For GA, initially, we form a randomly generated sequence, representing a page mapping. At each successive generation, the superior sequences in terms of vulnerability are selected as the evolutionary page mappings through the simulations. In order to generate the next sequence, 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



(a) Vulnerability Reduction (A bar greater than 1.0× indicates vulnerability reduction)



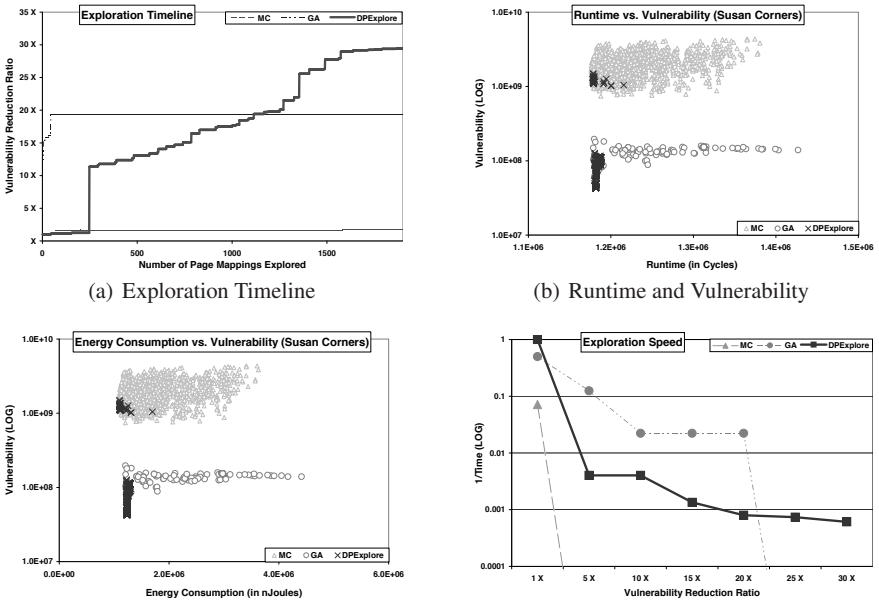
(b) Energy and Runtime Increase (A bar lower than 1.0× indicates the overhead)

Fig. 6 Evaluation under *5% Performance Penalty*: DPExplore significantly reduces the vulnerability at minimal runtime and power overheads

To demonstrate the effectiveness of DPExplore, we find the page partition with the least vulnerability under 5% performance penalty and exploration width 2. Fig. 6(a) plots the vulnerability ratio. *Vulnerability Ratio* indicates the ratio of the vulnerability of the *base case* to that 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. We observe 47× 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 is 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 Comparison with Other Explorations



(a) Exploration Timeline

(b) Runtime and Vulnerability

(c) Energy Consumption and Vulnerability

(d) Exploration Speed

Fig. 7 Exploration by MC, GA and DPExplore: DPExplore effectively explores 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. 7(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. 7(b) and Fig. 7(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. 7(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. However, the challenge is in partitioning pages among the two caches in a PPC. While page partitioning schemes have been proposed for multimedia applications, there is no page partitioning scheme for general applications. The page partitioning space is huge, and existing random techniques are unable to identify and explore the page partitions that lead to low vulnerability. In this paper, we develop DPExplore, a page partitioning algorithm at design time that effectively and efficiently finds page partitions resulting 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 reliability of cache-based architectures.

Our future work includes intelligent schemes to improve the data partitioning in PPCs for the varying input data at runtime, and partitioning techniques for instruction PPC caches.

References

1. *International Technology Roadmap for Semiconductors 2005 Executive Summary*. http://www.itrs.net/Links/2005ITRS/ExecSum2005.pdf.
2. L. Anghel and M. Nicolaidis. Cost reduction and evaluation of a temporary faults detecting technique. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages

591–597, 2000.
3. Ghazanfar-Hossein Asadi, Vilas Sridharan, Mehdi B. Tahoori, and David Kaeli. Balancing performance and reliability in the memory hierarchy. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 269–279, 2005.
4. Robert Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, pages 258–266, 2005.
5. Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.
6. D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 7–13, 2003.
7. J. Gaisler. Evaluation of a 32-bit microprocessor with builtin concurrent error-detection. In *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1997.
8. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop on Workload Characterization*, pages 3–14, 2001.
9. 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.
10. Hewlett Packard, http://www.hp.com. *HP iPAQ h4000 Series - System Specifications.*
11. Soontae Kim. Area-efficient error protection for caches. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 1282–1287, Mar 2006.
12. S. Krishnamohan and N. R. Mahapatra. An efficient error-masking technique for improving the soft-error robustness of static CMOS circuits. In *IEEE International SOC Conference (SOCC)*, pages 227–230, Sep 2004.
13. Kyoungwoo Lee, Aviral Shrivastava, Ilya Issenin, Nikil Dutt, and Nalini Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 411–420, Oct 2006.
14. Jin-Fu Li and Yu-Jane Huang. An error detection and correction scheme for RAMs with partial-write function. In *IEEE International Workshop on Memory Technology, Design and Testing (MTDT)*, pages 115–120, 2005.
15. Lin Li, Vijay Degalahal, N. Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Soft error and energy consumption interactions: A data cache perspective. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 132–137, Aug 2004.
16. P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1994.
17. Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, Feb 2005.
18. Kartik Mohanram and Nur A. Touba. Partial error masking to reduce soft error failure rate in logic circuits. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 433–440, 2003.
19. K. Mohr and L. Clark. Delay and area efficient first-level cache soft error detection and correction. In *IEEE International Conference on Computer Design (ICCD)*, 2006.
20. Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 37–42, 2004.
21. Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 29–40, Dec 2003.
22. M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *IEEE VLSI Test Symposium (VTS)*, page 86, 1999.

23. A. K. Nieuwland, S. Jasarevic, and G. Jerin. Combinational logic soft error analysis and protection. In *IEEE International Symposium on On-Line Testing (IOLTS)*, pages 99–104, 2006.
24. Richard Phelan. Addressing soft errors in ARM core-based designs. Technical report, ARM, 2003.
25. D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996. ISBN 0-1305-7887-8.
26. Nhon Quach. High availability and reliability in the Itanium processor. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 61–69, Sep–Oct 2000.
27. P. Shivakumar and N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. In *WRL Technical Report 2001/2*, 2001.
28. 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 *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 389–398, 2002.
29. Aviral Shrivastava, Ilya Issenin, and Nikil Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 90–96, 2005.
30. Makoto Sugihara, Tohru Ishihara, and Kazuaki Murakami. Task scheduling for reliable cache architectures of multiprocessor systems. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 1490–1495, 2007.
31. Synopsys Inc., Mountain View, CA, USA. *Design Compiler Reference Manual*, 2001.