Vulnerability-aware Dynamic Reconfiguration of Partially Protected Caches

Yuanwen Huang and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120, USA

***Abstract*—Cache vulnerability is a serious design concern due to exponential increase in soft errors with technology scaling. Partially Protected Caches (PPC) is a promising solution to mitigate vulnerability to soft errors in resource-constrained embedded systems. However, PPC suffers from both performance and energy overhead. Dynamic Cache Reconfiguration (DCR) is widely used in embedded systems to save energy and improve performance [13]. In this paper, we propose a methodology which takes advantage of the protected cache to reduce vulnerability, while utilizes reconfigurability to explore the trade-off between vulnerability, energy and performance. Experimental results demonstrate that our proposed method can significantly reduce both vulnerability (up to 87%) and energy consumption (up to 41%) without affecting the performance.**

I. Introduction

Soft errors, or transient faults induced by radiation, are becoming a major challenge in embedded systems design. When a high energy radiation particle strikes the diffusion region of a CMOS transistor and produces enough charge, it can flip the logic state of the transistor and cause a soft error. The cache in embedded microprocessors is most susceptible to the threat of soft errors [1], [2], [8] for several reasons: (i) cache occupies the majority of chip area, (ii) cache has an extremely high density of transistors, and (iii) cache cell size shrinks as technology scales down, which reduces the critical charge needed to flip a bit in stored data. A soft error in the cache corrupts the data, or changes the architectural state of a processor, which may eventually cause an observable difference in the behavior of the program and result in a failure. Consequently, it is very important to protect embedded caches from soft errors.

Several microarchitectural techniques have been proposed to reduce the vulnerability of memory data due to soft errors. Lee et al. [1] proposed the partially protected cache (PPC) architecture to mitigate cache vulnerability due to soft errors. The PPC architecture has two caches, one smaller cache that is protected, and the other unprotected cache which is prone to soft errors. The intuition behind PPC is that not all data is equally vulnerable to soft errors. By properly partitioning the data and mapping vulnerable data into the protected cache, PPC can significantly reduce the vulnerability or failure rate due to soft errors [6]. However, PPC is expensive in terms of energy consumption and performance, compared to the original architecture with only an unprotected cache.

In this paper, we propose a reconfigurable cache architecture (as shown in Figure 1) to address the above challenges. While

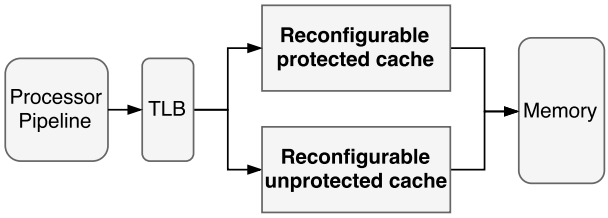


Fig. 1: A reconfigurable PPC-base architecture with one protected cache and the other unprotected cache at the same level of hierarchy.

the protected cache reduces vulnerability, the reconfigurability of the two (protected and unprotected) caches enables reduction in both execution time and energy consumption. This paper makes four important contributions: (i) it presents a reconfigurable cache architecture to improve performance and energy efficiency while maintaining PPC's natural advantage for vulnerability reduction; (ii) it develops a heuristic algorithm for partitioning data pages between the protected and unprotected caches; (iii) it proposes synergistic exploration of cache configurations and data partitioning schemes to trade-off between vulnerability, energy and performance; (iv) it proposes a fast exploration technique for selecting Pareto-optimal cache configurations.

The remainder of the paper is organized as follows. We discuss related works in Section II. Section III presents the energy and vulnerability models. Section IV describes the data partitioning method followed by our vulnerability-aware cache reconfiguration framework. Section V presents our experimental results. Finally, Section VI concludes the paper.

II. Related Work

We present related works in two separate categories: partially protected caches and dynamic cache reconfiguration.

A. Partially Protected Caches

The idea of Partially Protected Caches (PPC) initially comes from horizontally partitioned caches [7], where a processor has two or more caches at the same level of the memory hierarchy. Lee et al. [1], [6] extends the idea of horizontally partitioned caches into the PPC architecture, by assuming that one of the two caches is protected from soft errors. The protected cache has redundancy logic like SEC-DED [9] (single-bit error correction and double-bit error detection), which has overhead in access time, area and power consumption [10],

[11]. The challenge of using PPC is to properly partition data into the two caches to ensure that it would not introduce too much penalty in performance and energy consumption. However, the existing approaches [1], [6] use PPC only for the purpose of reducing vulnerability, and their approaches introduce unacceptable energy overhead.

B. Dynamic Cache Reconfiguration

Dynamic Cache Reconfiguration (DCR) is widely used for energy and performance optimization in embedded systems [12], [14]. There are many prior efforts in DCR to explore the trade-off between energy and performance. Wang et al. [12] presented a scheduling-aware cache reconfiguration for energy saving in real-time systems. Cai et al. [15] showed that different cache sizes could impact performance, energy and reliability. Huang et al. [2], [4] proposed a scheduling algorithm to reduce cache vulnerability using static profiling of a task set. The reconfigurable cache architecture chooses between different cache configurations depending on whether the goal is for energy optimization or vulnerability optimization. More recent work externed the idea of DCR for cache vulnerability reduction for multicore systems [3], and with machine learning [5]. None of these approaches consider PPC architecture. Our proposed approach combines the advantages of PPC and DCR through efficient data partitioning and cache exploration to improve both vulnerability and energy efficiency.

III. Energy / Vulnerability Models

In this section, we introduce the models used to measure energy consumption and vulnerability of the PPC caches.

Energy Model for the Unprotected Cache: The energy model for unprotected cache is adopted from the one used in [12]. The total cache energy consumption is $E = E_{dyn} + E_{sta}$, where E_{dyn} and E_{sta} denote the dynamic and static energy of the cache subsystem. Let E_{access} and E_{miss} be the energy consumption for per access and per miss, P_{sta} be the power consumption for one clock cycle (CC). Specially, we have:

$$E_{dyn} = Accesses \times E_{access} + Misses \times E_{miss} \quad (1)$$

$$E_{miss} = E_{offchip\_access} + E_{block\_fill} \quad (2)$$

$$E_{sta} = P_{sta} \times CC \times t_{cycle} \quad (3)$$

Energy Model for the Protected Cache: For the ECC protected cache, the dynamic energy calculation also includes energy consumption for ECC encode/decode. Similar to [1], we categorize the cache accesses into $read\_hit$, $read\_miss$, $write\_hit$, and $write\_miss$ since each operation results in different ECC events. For example, the energy consumption of $read\_hit$ is the sum of the access energy consumption and the energy consumption of ECC decoding (d), while the energy consumption of $read\_miss$ is the sum of the access energy consumption, the energy consumption of ECC decoding (d) as well as the ECC encoding (e).

$$\Delta E_{dyn} = RH \times d + RM \times (d + e) + WH \times e + WM \times (d + e) \quad (4)$$

where RH, RM, WH and WM denote the number of $read\_hit$, $read\_miss$, $write\_hit$ and $write\_miss$, respectively.

Vulnerability Model: Vulnerability analysis divides the lifetime of a piece of data into vulnerable and un-vulnerable intervals. Similar to [2], we measure the vulnerability of cache on a per-byte basis.

$$Vulnerability = \sum_{all\ bytes} vulnerable\ time\ of\ byte_i$$

A byte is vulnerable in an interval only if soft errors happen during the interval to contaminate the data (byte) and the contaminated data is either used by instructions (program) or written back to memory. Activities during the lifetime of a byte includes "idle", "fill", "read", "write" and "evict". The vulnerable intervals are of four types: fill-to-read, read-to-read, write-to-read, and write-to-evict. We measure the vulnerability of the unprotected cache as the summation of vulnerable intervals of all bytes in all cache blocks. When we profile an application to evaluate the vulnerability of each data page, the vulnerability of that page is the summation of vulnerability of all data that belongs to that page.

IV. Cache Reconfiguration of PPC

There are multiple aspects that have impact on a program executing on a reconfigurable PPC-based architecture. These aspects include: (1) the data page map which partitions data pages into the two caches; (2) the configuration of the protected cache and the configuration of the unprotected cache. Our goal is to minimize both vulnerability and energy consumption with acceptable or no degradation on performance. Since different programs have various data access patterns and cache requirements, we need to make wise design decisions to take advantage of the protected cache to reduce vulnerability, while utilize reconfigurability to save energy consumption.

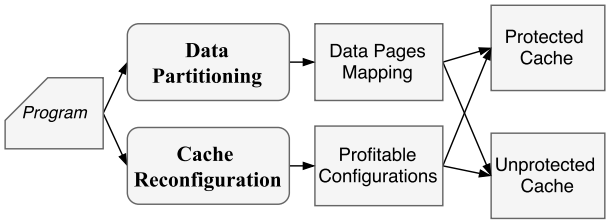


Fig. 2: Our exploration methodology consists of two design decisions: data partitioning and cache reconfiguration.

Figure 2 outlines our approach to accomplish the goal of vulnerability and energy optimization without penalizing the performance. The two important design decisions we have to make for each program include: (1) data partitioning to map data pages into the protected and unprotected caches; (2) cache reconfiguration to select the profitable configurations for the two caches. We perform these two steps through off-line (design time) analysis. It is important to note that off-line exploration is applicable and useful for many embedded systems because these systems have well-defined applications (programs) that are known a priori. The remainder of this section describes data partitioning and cache reconfiguration in detail.

A. Data Partitioning

The protected cache is very effective in reducing the vulnerability. For any data mapped to the protected cache, it is protected against soft errors. If we map all data into the protected cache, the vulnerability of the application will be reduced to zero. However, mapping too much data into the small protected cache is likely to increase the cache misses and eventually result in a significant degradation of performance and increase in energy consumption. We performed a simple experiment to show the effectiveness of protected cache in reducing vulnerability, as well as its side-effect on runtime and energy consumption. Figure 3 illustrate this exploration for benchmark $cjpeg$ from MediaBench [18]. First, we map all the data pages (54 pages in total for $cjpeg$) into the unprotected cache. Then we sort the pages by vulnerability in decreasing order. Each time we map a new page (from the top of the sorted list) into the protected cache if it would reduce the vulnerability. Figure 3a shows that after mapping 35 pages into the protected cache, vulnerability is reduced by 55%, while runtime increases by 10%. The runtime is expected to increase because the protected cache, which is much smaller in capacity, will cause a lot of misses when there are many data pages evicting each other's data. Figure 3b shows that energy consumption of unprotected cache decreases as we remove pages from the unprotected cache. However, the energy consumption of the protected cache will increase drastically when more data pages are mapped into it. This example suggests that we cannot afford to blindly map data pages into the protected cache, which might result in unacceptable performance and energy penalty.

Algorithm 1: DataPartition

Input: Benchmark, $rThresh$, page vulnerability profile
Output: $PageMap$

```
1  Set PageMap[n] = (0, 0, ..., 0);
2  {runtime, vulnerability} = simulate(PageMap)
3  Set BaseRuntime = runtime;
4  Set BestVulnerability = vulnerability;
5  Sort the pages by vulnerability in descending order
6  for (i = 0; i < PageMap.size; i++) do
7      Set PageMap[i] = 1;
8      {runtime, vulnerability} = simulate(PageMap)
9      if vulnerability < BestVulnerability then
10         if runtime < BaseRuntime × (1 + rThresh)
           then
11             Set BestVulnerability = vulnerability;
12         else
13             Set PageMap[i] = 0;
14     else
15         Set PageMap[i] = 0;
16 return PageMap
```

We introduce a greedy approach with a runtime threshold during data partitioning. For the same example above, if we set a 5% threshold for runtime, we are allowed to map only three pages into the protected cache. The runtime threshold will limit both the performance degradation and the energy penalty. Algorithm 1 shows our data partitioning approach. It takes the benchmark and the runtime penalty threshold ($rThresh$) as inputs, and produces a data partitioning scheme $PageMap$ as output. $PageMap[i]$ indicates the cache for the $i^{th}Page$: 0 means the unprotected cache, and 1 means the protected cache. We first map all pages into the unprotected cache (the base partitioning scheme) by setting $PageMap$ as all 0's (line 1). We simulate the benchmark with the base partitioning scheme, which provides us with $BaseRuntime$ and a profile of vulnerability of all data pages (line 2-4). After sorting the pages by vulnerability in descending order (line 5), we greedily select each page to test whether it is suitable to be mapped into the protected cache (line 6 to 15). A page is suitable to be protected if it satisfies two conditions: (i) it is favorable for vulnerability reduction, and (ii) it would not cause the program to exceed the runtime threshold. For the benchmarks used in our experiments, we have up to 259 pages, on average 58 pages. Our algorithm is very efficient with time complexity of $O(p)$, where p is the total number of data pages in the benchmark.

B. Cache Reconfiguration

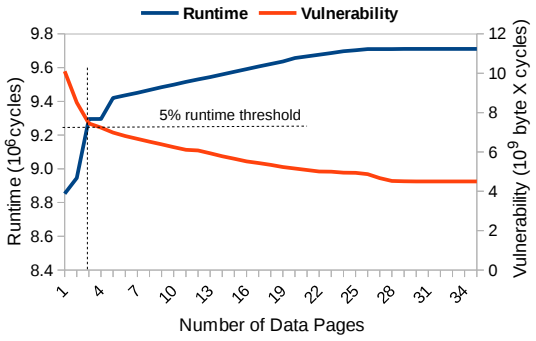
This section describes how to take advantages of both cache reconfiguration (DCR) and data partitioning (PPC) by synergistic exploration. In our reconfigurable PPC architecture, the **base cache**[1] for the unprotected cache is 4096B_1W_32B (size of 4KB, 1-way associative, and line size of 16-byte), which can be reconfigured to the size of 1KB, 2KB and 4KB, associativity of 1-way, 2-way and 4-way, line size of 16-byte, 32-byte and 64-byte. This will lead to 18 valid configurations[2] for the unprotected cache. The base configuration for the protected cache is 512B_1W_32B, which can be reconfigured to the size of 256B and 512B, associativity of 1-way and 2-way, and line size of 16-byte and 32-byte. This will lead to 6 valid configurations for the protected cache.

It is crucial to dynamically select partitioning schemes for different cache configurations. For example, assume that we have a data partitioning solution for the base configuration (<512B, 4096B>) for the protected and unprotected caches. If we use the same data partitioning for another cache configuration (<256B, 4096B>), it would likely encounter a lot of cache misses in the protected cache and result in significant performance and energy penalty.

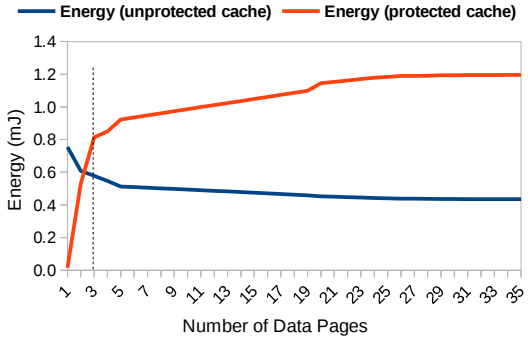
The effectiveness of the data partitioning algorithm, described in Section IV-A, is influenced by the cache exploration. First, the data partitioning depends on the configuration of the unprotected cache, as it requires the vulnerability profile of data pages. When using a different unprotected cache, the vulnerability of a data page will change, which will directly affect its priority (importance in reducing vulnerability) during

[1]The **base cache** refers to the cache configuration that is widely used in the literature for the selected benchmarks.

[2]It is not equal to 3^3 since not all combinations are valid [12].



(a) Vulnerability and Runtime trade-off



(b) Energy Consumption

Fig. 3: Trade-off between vulnerability, runtime and energy, for benchmark $cjpeg$. As more pages are mapped into the protected cache, vulnerability goes down rapidly, while runtime and the total energy consumption increase significantly.

data partitioning. Similarly, the data partitioning is a greedy algorithm and is constrained by the configuration of the protected cache. We propose a synergistic exploration of different cache configurations with various partitioning of data pages.

Algorithm 2: CacheReconfiguration

Input: Benchmark, $rThresh$
Output: Protected and unprotected cache configs

```
1  for cache size s_u of 1024B, 2048B, 4096B do
2      for associativity a_u of 1, 2, 4 ways do
3          for line size l_u of 16B, 32B, 64B do
4              UnproConfig = (s_u, a_u, l_u)
5              Generate vulnerability profile
6              for cache size s_p of 256B, 512B do
7                  for associativity a_p of 1, 2 ways do
8                      for line size l_p of 16B, 32B do
9                          ProConfig = (s_p, a_p, l_p)
                           // Call Algorithm 1
10                         DataPartition(rThresh)
11 Collect (runtime, vulnerability, energy) for all configs.
12 Choose the best configs based on system goal.
13 return (UnproConfig, ProConfig, PageMap)
```

Algorithm 2 shows our approach of dynamic data partitioning during the process of cache exploration. We explore the cache size, way associativity, and line size of the unprotected cache in line 1-3. For each unprotected cache configuration, it is necessary to re-evaluate the vulnerability of all pages based on the current configuration of the unprotected cache (line 4-5). Next, we explore the configurations of the protected cache and call Algorithm 1 to get the best data partitioning for the selected pair of cache configurations (line 6-10). The complexity of the algorithm is $O(n_1 \times n_2 \times p)$, where n_1 and n_2 are the number of configurations for the unprotected and protected caches, respectively, and p is the number of data pages in the benchmark.

C. Fast Exploration of Caches

One major drawback of the exhaustive approach outlined in the previous section is the long exploration time. The total time for exhaustive exploration is $(C \times n_1 \times n_2 \times p)$. For the largest benchmark $epic$ with p=258 pages, the time needed for one simulation is C=22 seconds, so the total time is (22×18×6×258) seconds, which is about 7 days. In order to improve the scalability of our reconfiguration framework, we propose a fast and effective exploration heuristics that would explore fewer cache configurations without compromising the quality of optimization objectives.

By examing the results generated by exhaustive exploration, we find that some very unprofitable cache configurations are also explored. We have 18 configurations for the unprotected cache and 6 configurations for the protected one. Both caches (protected and unprotected) have influence on the overall vulnerability and energy consumption. If one configuration for the protected (or unprotected) cache is very bad, it may not be useful to explore the unprotected (or protected) cache at all. We can explore the two caches independently and pick out the Pareto-optimal tradeoff points. Candidates with both vulnerability and energy consumption worse than the Pareto-optimal ones are eliminated during the exploration. Our proposed Fast Exploration of Caches (FEC) heuristic is summarized below:

1) Hold the protected cache as the base configuration. Tune the unprotected cache and record all its Pareto-optimal configurations. Let P_u denote the set of recorded configurations for the unprotected cache.
2) Hold the unprotected cache as the base configuration. Tune the protected cache and record all its Pareto-optimal configurations. Let P_p denote the set of recorded configurations for the protected cache.
3) Explore all the combinations from each set of Pareto-optimal configurations from the previous two steps, and find the best combination for the protected and unprotected caches.

In the last step, we choose the primary optimization objective as the *energy-aware vulnerability-minimization* exploration in our experiments. In other words, our exploration finds

the best cache configurations with lowest vulnerability while the energy consumption is equal or better than two (protected and unprotected) fixed base caches (no DCR capability).

The first two steps explore 24 (=18+6) candidates while the last step explores $|P_u|*|P_p|$ candidates. The number of Pareto-optimal points varies for different applications but normally around 2 to 6. In our experiments, the total number of explored cache configurations is 35 on average for the above heuristic approach. The total exploration time for one benchmark will be reduced to $C \times n' \times p$, where $n' = 35$ on average. The number of cache configurations is reduced from 108(=18*6) to 35, which results in about 3X speed-up. Our experiments show that our fast exploration approach can find the best cache combination without compromising the quality of optimization objectives.

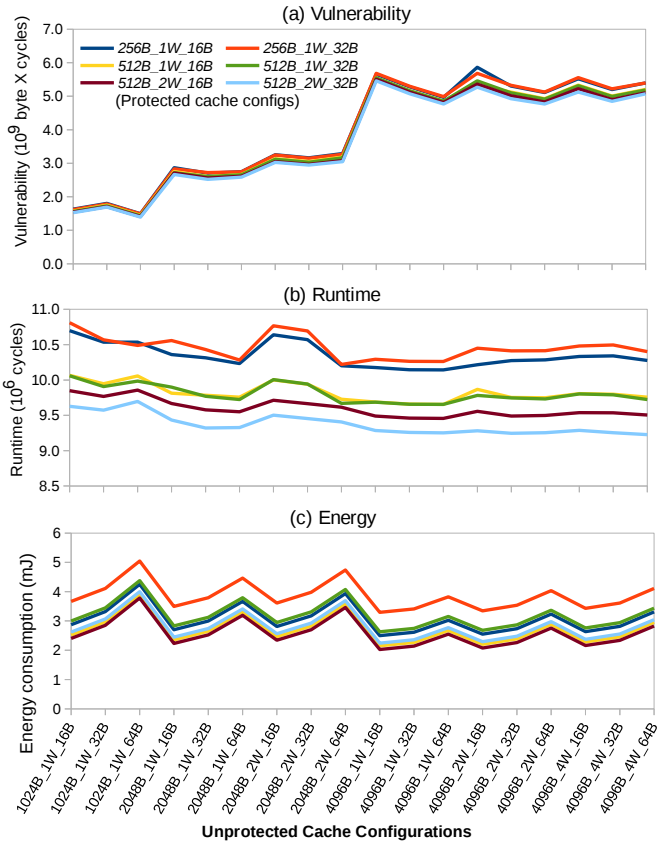


Fig. 4: Exploration of all cache configurations for benchmark $cjpeg$, with $rThresh = 5\%$.

V. Experiments

Our framework used the *sim-outorder* simulator from the SimpleScalar toolset [16]. The protected cache has an ECC-based technique, while the unprotected cache has no protection against soft errors. We assume that the protected cache is optimized to have the same access time as the unprotected one. Both caches are reconfigurable, and the reconfigurable cache model is described in Section III. The ten benchmarks that are used in our experiments are from the MediaBench [18] and MiBench [19], which are representative of embedded system applications. The energy model and vulnerability model are detailed in Section III. The energy consumption for cache accesses is estimated using CACTI 4.2 [17] with a 0.18 μm technology. We implemented the vulnerability calculation in the simulator for the unprotected cache.

A. Synergistic Exploration

In this subsection, we explore the interesting trade-off between vulnerability, runtime, and energy consumption.Figure 4 shows the exhaustive exploration on 18*6 cache configurations for benchmark $cjpeg$, with runtime penalty threshold to be 5%.

Figure 4(a) highlights two interesting observations about vulnerability, which are also observed for other benchmarks. (1) Vulnerability is dominated by the unprotected cache. For different protected caches with same unprotected cache, the vulnerability is almost the same. This is as expected, since vulnerability only comes from data maintained in the unprotected cache. (2) The vulnerability of the first nine unprotected cache configurations (of size 1024B and 2048B) is smaller than that of the last nine configurations (of size 4096B). This is due to the fact that a large unprotected cache can retain more data and the data usually stay in the cache for longer time because of lower miss rates, compared with a small unprotected cache.

Figure 4(b) shows the runtime of $cjpeg$. For this benchmark, the upper two curves (protected cache of a small size of 256B) have very bad runtime, compared with the other four configurations of size 512B. This is because $cjpeg$ is a data-intensive benchmark, and the most vulnerable pages happen to be the most frequently accessed pages. A small size (256B) cache can protect data from vulnerability while incurring a lot of misses and causing performance degradation. Figure 4(c) shows the energy consumption of $cjpeg$. For this benchmark, the curve for the protected cache ($256B\_1W\_32B$) has much worse energy consumption than others, and this is also related to its long runtime. For a fixed protected cache, the energy consumption fluctuates because of different configurations of the unprotected cache.

TABLE I: Vulnerability reduction compared to DCR [2]

Benchmark	DCR [2]	Our Approach	**Improvement**
fft	2.55E+10	1.19E+09	**95.3%**
cjpeg	1.01E+10	3.78E+09	**62.6%**
djpeg	1.95E+09	1.68E+09	**13.6%**
pegwit	2.26E+09	2.46E+08	**89.1%**
rijndael	6.78E+10	6.78E+10	**0.0%**
stringsearch	2.02E+09	6.70E+08	**66.8%**
untoast	2.46E+10	1.53E+09	**93.8%**
epic	6.44E+09	4.30E+09	**33.2%**
ospf	2.42E+09	2.89E+08	**88.0%**
susan	9.82E+09	6.36E+09	**35.2%**
Average	-	-	**57.8%**

B. Comparison with Previous Works

In this subsection, we compare our results with [2] and [6] to demonstrate the effectiveness of our approach.

(1) Comparison of our approach with DCR [2]:

The approach in [2] proposed to use dynamic cache recon-

TABLE II: Comprison of our DCR+PPC approach with PPC [6]

Benchmark	**Vulnerability** ($10^9 byte \times cycles$)			**Energy** (mJ)			**Runtime** ($10^6 cycles$)		
	PPC [6]	Our Approach	**Improvement**	PPC [6]	Our Approach	**Improvement**	PPC [6]	Our Approach	**Improvement**
fft	3.50	1.19	**66.1%**	10.1	5.98	**40.5%**	59.9	58.3	**2.7%**
cjpeg	6.33	3.78	**40.3%**	2.14	2.08	**2.7%**	9.22	9.54	**-3.4%**
djpeg	3.36	1.68	**49.9%**	0.34	0.32	**6.8%**	2.63	2.69	**-2.1%**
pegwit	0.86	0.25	**71.3%**	4.42	4.30	**2.6%**	17.5	17.6	**-0.7%**
rijndael	80.1	67.8	**15.4%**	3.84	3.20	**16.7%**	46.3	46.3	**0.0%**
stringsearch	2.92	0.67	**77.1%**	0.66	0.62	**5.2%**	6.81	6.69	**1.9%**
untoast	1.61	1.53	**5.0%**	2.60	2.45	**5.6%**	29.5	29.3	**0.9%**
epic	33.4	4.30	**87.1%**	4.80	3.84	**19.9%**	36.2	37.5	**-3.6%**
ospf	0.76	0.29	**62.1%**	1.54	1.20	**22.1%**	5.19	5.06	**2.5%**
susan	15.4	6.36	**58.7%**	1.05	1.03	**1.8%**	15.8	15.2	**3.5%**
Average	-	-	**53.3%**	-	-	**12.4%**	-	-	**0.2%**

figuration (DCR without PPC) to reduce vulnerability. Table I compares our DCR+PPC approach with them. The results for DCR [2] are from the EAVO approach in [2], which uses only one reconfigurable cache. Our approach can reduce vulnerability by up to 95.3%, on average 57.8%, for the ten benchmarks. For $rijndael$, our vulnerability number is the same as [2]. This is because our DCR+PPC configuration is the same as DCR, which means we map all the data into the unprotected cache while the protected one is not used at all. The reason is that $rijndael$ has only 17 data pages (the smallest among all benchmarks), and mapping any of the data pages into the protected cache resulted in exceeding the performance threshold. Our approach is able to provide drastic improvement in vulnerability compared to [2], because we take advantage of PPC's ability to reduce vulnerability.

(2) Comparison of our approach with PPC [6]:

Table II shows the detailed results. The first column indicates the benchmark. The second, fifth and eighth columns provide the vulnerability, energy and execution time, respectively, for the base configuration with PPC [6]. The third, sixth and ninth columns provide vulnerability, energy and execution time, respectively, using our approach (DCR+PPC). The fourth, seventh and tenth columns indicate the improvement produced by our approach compared to [6]. A positive number implies improvement whereas a negative number means overhead (penalty). Table II shows that our approach can provide significant reduction (53.3% on average) in vulnerability, modest reduction (12.4% on average) in energy, and minor performance improvement (0.2% on average).

Anthor interesting obsevation is that we have very minor (at most 3.6%) performance penalty. Although we set the runtime threshold to be 5% in our data partitioning algorithm, all benchmarks have far better performance than the threshold. In fact, many of them even have performance improvement. The is due to the fact that although PPC has the potential to cause performance degradation, DCR can find the suitable cache configurations to hide the performance penalty. This further demonstrates the effectiveness of our DCR+PPC approach.

VI. Conclusion

Designing reliable embedded systems needs to consider cache vulnerability due to soft errors. Partially protected caches (PPC) provide an effective mechanism to reduce cache vulnerability. However, it may introduce unacceptable energy and performance overhead. In this paper, we presented a reconfigurable cache architecture to combine the advantages of PPC (vulnerability reduction) and cache reconfiguration (energy and performance improvement). Synergistic integration of cache reconfiguration and data partitioning improved both vulnerability and energy efficiency. Our vulnerability-aware energy minimization approach can significantly reduce vulnerability (up to 87.1% and on average 53.3%) as well as energy (up to 40.5% and on average 12.4%) without affecting the performance.

References

[1] K. Lee et al. Mitigating soft error failures for multimedia applications by selective data protection. CASES, 2006.

[2] Y. Huang and P. Mishra. Reliability and Energy-aware Cache Reconfiguration for Embedded Systems. ISQED, 2016.

[3] Y. Huang and P. Mishra. "Vulnerability-aware Energy Optimization using Reconfigurable Caches in Multicore Systems," ICCD, 2017.

[4] Y. Huang and P. Mishra. "Vulnerability-aware Energy Optimization for Reconfigurable Caches in Multitasking Systems," TCAD, 2018.

[5] A. Ahmed, Y. Huang and P. Mishra. "Cache Reconfiguration using Machine Learning for Vulnerability-aware Energy Optimization," TECS, 2019.

[6] K. Lee et al. Partitioning Techniques for Partially Protected Caches in Resource-Constrained Embedded Systems. TODAES, 2010.

[7] A. Shrivastava et. al. Compilation techniques for energy reduction in horizontally partitioned cache architectures. CASES, 2005.

[8] M. M. Sabry et al. OCEAN: An Optimized HW/SW Reliability Mitigation Approach for Scratchpad Memories in Real-Time SoCs. TECS, 2014.

[9] N. N. Sadler and D. J. Sorin. Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache. ICCD, 2006.

[10] J.-F. Li and Y.-J. Huang. An Error Detection and Correction Scheme for RAMs with Partial-Write Function. MTDT, 2005.

[11] Doe Hyun Yoon and Mattan Erez. Memory mapped ECC: low-cost error protection for last level caches. SIGARCH Comput. Archit., 2009.

[12] W. Wang et al. Dynamic Cache Reconfiguration for Soft Real-Time Systems. TECS, 2012.

[13] W. Wang et al. Dynamic Reconfiguration in Real-Time Systems - Energy, Performance, Reliability and Thermal Perspectives. Springer, 2012.

[14] P. Hsu and T. Hwang. Thread-criticality aware dynamic cache reconfiguration in multi-core system. ICCAD, 2013.

[15] Y. Cai et al. Cache size selection for performance, energy and reliability of time-constrained systems. ASP-DAC, 2006.

[16] T. Austin et al. SimpleScalar: An Infrastructure for Computer System Modeling. Computer, 2002.

[17] T. David et al. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[18] Lee, Chunho et al. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. IEEE Computer Society, 1997.

[19] Guthaus, Matthew R., et al. MiBench: A free, commercially representative embedded benchmark suite. WWC, 2001.