Vulnerability-aware Energy Optimization for Reconfigurable Caches in Multitasking Systems

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

***Abstract*—Cache vulnerability due to soft errors is one of the reliability concerns in embedded systems. Dynamic reconfiguration techniques are widely studied for improving cache energy without considering the implications of cache vulnerability. Maintaining a useful data longer in the cache can be beneficial for energy improvement due to reduction in miss rates, however, longer data retention negatively impacts the vulnerability due to soft errors. This paper studies the trade-off between energy efficiency improvement and reduction in cache vulnerability during cache reconfiguration. We propose a heuristic approach for both inter-task and intra-task cache reconfiguration in multitasking systems. Experimental results demonstrate that our proposed approach can significantly improve both vulnerability (25% on average) and energy efficiency (21% on average) for data cache without violating real-time constraints.**

I. Introduction

Soft errors are transient faults in CMOS circuits, which are caused by energy carrying particles (cosmic rays or substrate alpha particles). These transient faults flip bits in storage cells or change the logic values in functional units. Soft error rate per chip is expected to grow due to the growing density of transistors on chip [2][17]. Previous studies have concluded that unprotected memory elements are the most vulnerable components to soft errors [3]. The cache in embedded microprocessors is most susceptible to soft errors 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 scales down, which reduces the critical charge needed to flip a bit in stored data. Due to widespread use of embedded systems in safety-critical devices, it is necessary to protect embedded caches from soft errors.

Dynamic Cache Reconfiguration (**DCR**) is a widely studied method for optimizing energy and performance in embedded systems [4]. The basic idea of cache reconfiguration is that different programs have varying data and instruction access characteristics during execution (runtime) and DCR tries to find the optimal cache configuration for a given application (program). For example, we can improve performance by increasing cache size when a program needs a lot of data accesses. Similarly, we can save energy by shutting down a part of the cache if the program is not data-intensive. However, cache reconfiguration will also affect the vulnerability due to soft errors. A large cache size for a data-intensive program might have fewer cache misses and thus improve energy and performance efficiency, but it is also likely to increase the vulnerability of cache data because of longer data retention in the cache. This interesting trade-off between performance, energy and vulnerability is the motivation for this work.

This work was partially supported by the NSF grant CNS-1526687. Previous conference version of the paper is in [1]. Corresponding author: Yuanwen Huang, *yuanwenhuang@ufl.edu*.

It is a major challenge to improve the reliability of real-time embedded systems with special design considerations of real-time constrains [6][13]. Hard real-time systems require that all tasks must complete execution before their deadlines to ensure correct execution. Due to stringent timing constraints, scheduling for hard real-time systems must perform task schedulability analysis based on task attributes [5]. For soft real-time systems, minor deadline misses may result in temporary service degradation, but will not lead to incorrect behavior. An efficient cache reconfiguration framework is proposed for energy optimization in soft real-time systems in [4]. They exploit the flexibility of soft real-time systems and manage to achieve considerable energy savings with minor impact on user experiences. However their method does not consider the vulnerability of cache due to soft errors.

To the best of our knowledge, there are no prior efforts in analyzing the cache vulnerability during cache reconfiguration. We propose a methodology for using cache reconfiguration in soft real-time systems. Our approach provides an efficient cache tuning strategy based on static profiling and dynamic scheduling of tasks. We explore Vulnerability-aware Energy Optimization (VAEO) opportunity within each task (*intra-task VAEO*) as well as across task sets (*inter-task VAEO*). While traditional approaches (no DCR) uses a fixed cache for all tasks in the system, the inter-task DCR will select (use) the most beneficial cache configuration for each task to improve both vulnerability and energy-efficiency. Intra-task DCR will extend the optimization opportunity further by enabling changes in cache configuration within each task. Our proposed research is able to balance performance, energy consumption and vulnerability, so that tasks can meet their deadlines and produce energy savings while vulnerability reduction can also be achieved.

The rest of the paper is organized as follows. Section II presents related work on DCR and cache vulnerability. Section III motivates the reader by illustrating the effect of DCR on performance, energy consumption and vulnerability. Section IV presents our cache reconfiguration methodology for inter-task VAEO. Section V presents intra-task VAEO, which includes phase identification and cache configuration assignment for phases. Section VI presents the experimental results. Finally, Section VII concludes the paper.

II. Background and Related Work

This section surveys existing works in two related domains: cache reconfiguration and cache vulnerability.

A. Cache Reconfiguration

Applications have varied instruction and data access patterns, which means that they require different cache requirements in terms of cache size, line size, and associativity. If the cache configuration is tuned according to the need of the application, we can gain performance improvement and energy savings. The configurable cache architecture used in our work is similar to the one in [7]. It contains four cache banks operating as four separate ways. The cache ways can be configured to shut down so as to vary the cache size. The way associativity can be changed by concatenating ways. The line size can be adjusted by configuring the fetch unit to different lengths. This architecture requires very simple hardware augmentation and minor overhead [7]. A light process can be used as the cache tuner, which will make the reconfiguration decision and change the configuration at runtime.

Figure 1 illustrates that inter-task and intra-task DCR can improve overall performance by tuning cache size for a system with three tasks. We assume that cache size is the only tunable parameter of cache for the ease of illustration (line size and associativity remain the same). Figure 1(a) shows a traditional system using a fixed *base cache*[1], whereas in Figure 1(b) each task uses its favorable cache configuration and the overall execution time is improved. In the traditional system with a fixed *base cache*, Task 1 starts to execute at time t_0, Task 2 and Task 3 start at t_1 and t_2 respectively. The fixed *base cache* has a 4096-byte cache size for all tasks. In inter-task (application-based) cache reconfiguration, DCR tunes the cache when a new task starts its execution. Assuming Task 1 is computation-intensive, we choose a smaller (2048-byte) cache to save energy, while the execution time will increase. Assuming Task 2 is data-intensive, we choose a larger (8192-byte) cache and its runtime is greatly improved. Figure 1(c) shows the effect of combining inter- and intra-task cache reconfiguration. By introducing intra-task reconfiguration, Task 1 can improve its performance if suitable configurations are applied to the four phases during execution. Assuming Task 2 has three phases, we set the cache to be large (8192-byte) for the first and third phase for performance consideration, and set the second phase to 4096-byte to reduce energy consumption. For Task 3, inter-task reconfiguration is not able to find a better cache than 4096-byte, while intra-task reconfiguration can find three phases and improve the performance and/or energy. For inter-task reconfiguration, the performance overhead is negligible since the processor needs to anyway switch context and start with a fresh cache [4][7][10]. For intra-task reconfiguration, the cache will be flushed if we decide that the configuration is to be changed for the new phase. The overhead of intra-task reconfiguration will be discussed further in Section V.

[1]***Base cache*** refers to the cache used in typical real-time systems, which is chosen to ensure durable task schedules. Typically, *base cache* is the globally optimal cache configuration determined during design time for a set of tasks.

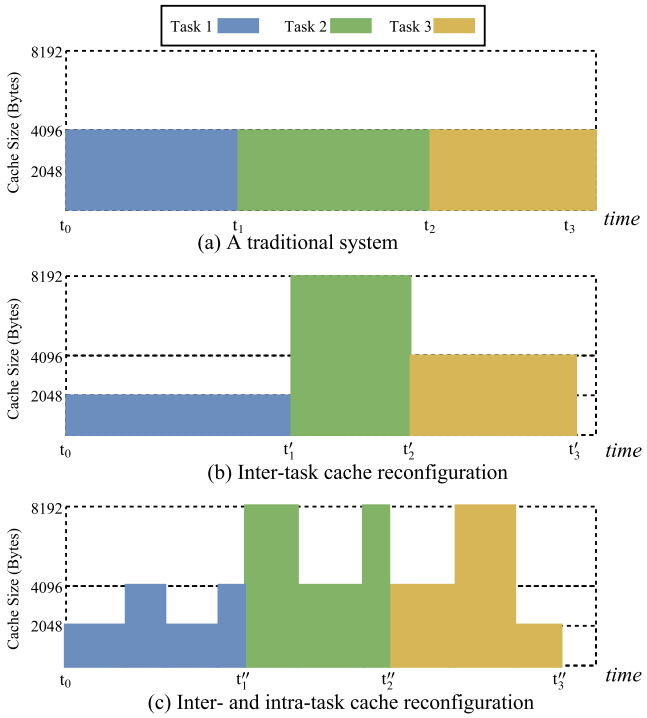


Fig. 1: DCR in a system with three tasks.

DCR has been extensively studied by previous works [4][7][9][10]. The configurable cache architecture used in our work is similar to the one in [7]. It contains four cache banks operating as four separate ways. The cache ways can be configured to shut down so as to vary the cache size. The way associativity can be changed by concatenating ways. The line size can be adjusted by configuring the fetch unit to different lengths. This architecture requires very simple hardware augmentation and minor overhead [7]. A light process can be used as the cache tuner, which will make the reconfiguration decision and change the configuration at runtime. There are many prior efforts in developing energy- and performance-aware cache reconfiguration techniques. Wang et al. [7] studied scheduling-aware cache reconfiguration for energy saving in real-time systems. [8] combined DCR with dynamic voltage scaling for leakage-aware energy minimization. Phase-level DCR [26][27][28] showed that intra-task reconfiguration can improve energy consumption along with inter-task reconfiguration. Cai et al. [11] showed that cache size could impact performance, energy and reliability. However, none of the previous works has considered cache vulnerability improvement during DCR. In this paper, we use both inter-task and intra-task reconfiguration to improve vulnerability and energy consumption.

B. Cache Vulnerability

In order to facilitate reliability analysis of cache, a measurement method is needed for the quantification of cache vulnerability due to soft errors [12]. Mukherjee et al. [15] introduced the concept of Architectural Vulnerability Factor (AVF). Vulnerability analysis divides a bit's lifetime into

vulnerable and un-vulnerable intervals [15][16]. A bit is vulnerable for an interval, if soft errors that happen in this interval will cause the program to get contaminated data. Similar to [18] and [21], we measure the vulnerability of cache on a per-byte basis. Activities during the lifetime of a byte includes "*idle*", "*read*", "*write*" and "*eviction*". Figure 2(a) shows a data with both *read* and *write* accesses, and the vulnerable intervals are marked by two black rectangles: the data is vulnerable between the first *write* and the second *read* as well as between the second *write* and the *evict*. During these two intervals, the data needs to be read for reuse, while a flipped bit can corrupt the data, causing the program to use the corrupted cache data. The interval between the second *read* and the second *write* is un-vulnerable, since the data will be updated by the *write* operation even if soft errors corrupt it. Figure 2(b) shows a data with only *read* accesses, and the intervals between *read* accesses are vulnerable. However, the interval between the last *read* and the *evict* is un-vulnerable, since data will not be reused or written back to memory.

Byte Cycles is an widely used term for measuring cache vulnerability [3], [18]. We measure the vulnerability of cache as the summation of vulnerable intervals of all bytes. It can be defined as follows:

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

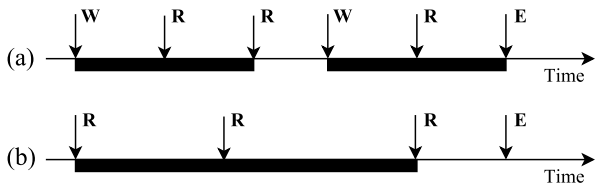
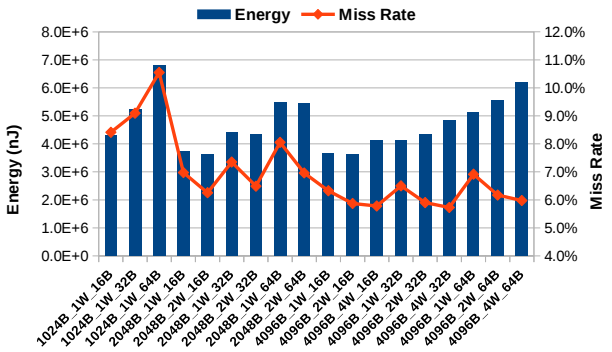


Fig. 2: Vulnerable intervals of two data elements in a cache without soft-error protection (where W=Write Access, R=Read Access, E=Evict). (a) data with both *write* and *read* accesses; (b) data with only *read* accesses.

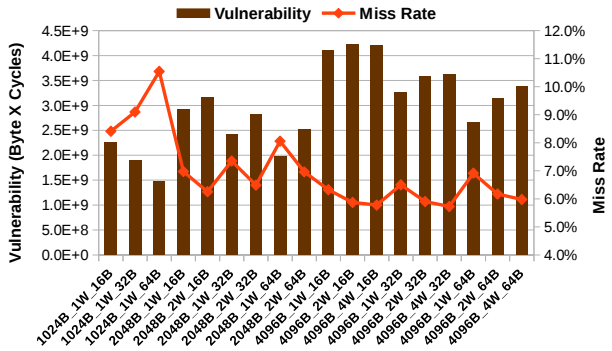
Major reliability improvement techniques include error correction and error prevention [2][14][19]. Error correction techniques, such as parity caching and error-correcting codes (ECC), use spatial redundancy to detect errors. If an error is detected in a cache block and this block is not dirty (i.e. memory has a correct copy of this block), it is possible to recover by reloading from memory. But if an error is detected in a dirty block, there is no way to recover the corrupted data. An important idea in protecting cache data from soft errors is to ensure that there is an updated copy of all cached data in memory (so data can be reloaded if soft error corrupts data). Error prevention techniques [20], such as periodic flushing and early write-back, are introduced. However, too many memory-writes will keep the data-bus busy, which results in longer cache-miss latency and decreased overall performance. Particularly, write-through caches will always write data all the way to memory, but may not be a good idea for embedded systems. Shrivastava et al. [30] provide analysis on cache vulnerability for programs which have a specific data access pattern (an n-level nested loop) for a directly-mapped cache. However, for programs with diverse data access patterns, their approach has very limited applicability.

For caches with error detection or correction techniques, the detection/correction process comes at a cost. It takes clock cycles to correct the error and the CPU might get stalled to re-fetch the data if the error cannot be successfully corrected by ECC. In this paper, we did not assume any error correction techniques for L1 caches in our paper. Our goal is to take advantage of the reconfigurable cache and the data access pattern of applications to reduce vulnerability and improve energy efficiency while meeting task deadlines. Our approach is orthogonal to error correction techniques. One obvious way to combine our proposed approach (VAEO) and ECC would be to use VAEO to reduce soft errors as much as possible and then apply ECC to detect/correct the remaining soft errors.

III. Motivation: Illustrative Example



(a) Energy and Miss Rate of *pegwit*



(b) Vulnerability and Miss Rate of *pegwit*

Fig. 3: Energy (a) and vulnerability (b) values of *pegwit* benchmark using different cache configurations. This figure is the same as the one in our conference paper [1].

Existing techniques for cache reconfiguration do not consider cache vulnerability due to soft errors. Figure 3 illustrates the interesting behaviors of vulnerability and energy consumption under different cache configurations. We run the program *pegwit* (a benchmark from MediaBench [22]) for 18 times, and each run uses a different configuration for L1 data cache. Each configuration consists of three parameters: cache size, associativity and line size. For example, 1024B_1W_64B

implies a cache configuration with cache size of 1024 bytes, one way with 64 bytes line size.

Figure 3 shows that the energy consumption, vulnerability and miss rate change drastically as we tune cache configurations. Both energy and vulnerability relate to cache miss rates and cache configurations. However, the correlation behaviors are quite different and even conflicting in certain scenarios. In Figure 3(a), energy consumption decreases when miss rate decreases (the first 9 cache configurations), but keeps increasing for the last 9 cache configurations even though miss rates are fairly low. The reason is that the total energy consumption is the sum of dynamic and static energy. For the first 9 cache configurations, the total energy is dominated by dynamic energy consumption, thus the total energy decreases when miss rate (dynamic energy consumption) decreases. However, for the last 9 cache configurations with large cache size, the total energy is dominated by static energy consumption even though miss rates are low. In Figure 3(b), the relation between vulnerability and miss rate is a little more complex. Cache size has a significant influence on vulnerability. Configurations with cache size of 1024B is much less vulnerable than configurations with cache size of 2048B and 4096B. For configurations with the same cache size, vulnerability decreases when miss rate increases and vice versa. For the same cache size, lower miss rate means that more dirty data is staying in cache for longer time, which contributes to vulnerability.

There are two interesting observations here: (i) small cache size might have high energy consumption but less vulnerable; (ii) low miss rate might be energy friendly but leads to higher vulnerability. These observations motivate us to investigate the trade-off between vulnerability, energy and performance during DCR. In this paper, we develop a cache reconfiguration framework that considers both energy and cache vulnerability. Since both vulnerability and energy depend on program characteristics and cache configurations, we statically analyze various cache configurations for each application. Such an approach is suitable for embedded systems since applications are known a priori. Based on static analysis, we propose inter-task as well as intra-task dynamic cache tuning that can select suitable configurations during runtime.

IV. Inter-task Cache Reconfiguration

A. System Model

Let us define the reliability-aware DCR problem with consideration of both energy and cache vulnerability. The system we consider can be modeled as:

- A processor with a reconfigurable cache which supports m possible cache configurations $C = \{c_1, c_2, c_3, ..., c_m\}$.
- A set of n independent tasks $T = \{t_1, t_2, t_3, ..., t_n\}$.
- Each task $t_i \in T$ has attributes including arrival time, period and deadline. Non-preemptive execution is employed, which means, a task will continue execution until completion once it starts to execute.

Let $e_{t_i}^{c_j}$, $p_{t_i}^{c_j}$ and $v_{t_i}^{c_j}$ denote the energy, execution time (performance) and vulnerability of task t_i when it is run on cache configuration c_j. The reliability-aware DCR problem is to find a cache assignment for the task set such that energy consumption and vulnerability are minimized with each of the tasks satisfying its deadline. One common practice for dealing with multi-objective optimization problem is to optimize one objective at a time while transforming other objectives into constraints. We introduce the *Vulnerability-aware Energy Optimization* (**VAEO**) problem, which aims at minimizing the total energy consumption, while adding vulnerability of tasks as constraints. A heuristic algorithm based on run-time task scheduling is proposed for solving the VAEO problem. Note that our proposed VAEO approach can be extended to solve the *Energy-aware Vulnerability Optimization* **EAVO** problem with simple augmentation.

B. Vulnerability-aware Energy Optimization (VAEO)

Let n represent the total number of task arrivals within the least common multiple (hyper-period[2]) of all task periods. $\sum_{i=1}^{n} e_{t_i}^{c_j}$ is the total energy consumption of n tasks[3]. The VAEO DCR problem can be defined as the following:

$$\text{minimize} \sum_{i=1}^{n} e_{t_i}^{c_j} \tag{1}$$

subject to

$$v_{t_i}^{c_j} \leq V_{t_i},\ \forall i \in [1, n] \tag{2}$$

$$a_{t_i} + w_{t_i} + p_{t_i}^{c_j} \leq D_{t_i},\ \forall i \in [1, n] \tag{3}$$

Equation 1 is the optimization objective. Equation 2 and 3 contain the vulnerability and timing constraints. V_{t_i} is the upper bound for vulnerability of task t_i. Here a_{t_i}, w_{t_i}, $p_{t_i}^{c_j}$, D_{t_i} denote the arrival time, queuing time, execution time, and deadline of task t_i. The optimization goal is to find a set of cache configuration assignments for all tasks so that the total energy consumption is minimized with vulnerability and timing constraints. We choose V_{t_i} as the vulnerability of task t_i when it is executed with the *base cache*, the most profitable cache configuration decided during design time. In other words, we set the vulnerability as a constraint to ensure that it is always at least as reliable as the *base cache*.

In Equation 3, arrival time a_{t_i} and deadline D_{t_i} are known upon the arrival of the task, while queuing time w_{t_i} and execution time $p_{t_i}^{c_j}$ depend on the scheduling and cache reconfiguration algorithms. Queuing time w_{t_i} depends on the scheduler and is determined by the priority of this task and the other tasks currently in the queue. Execution time $p_{t_i}^{c_j}$ is determined by the cache configuration c_j which will be assigned to this task by the cache reconfiguration algorithm.

C. Heuristic Approach for VAEO Problem

Tasks arrive periodically and each task is inserted into a list of ready tasks upon arrival. We propose a heuristic approach, which employs Earliest Deadline First (EDF) as our

[2]A hyper-period is the Least Common Multiple (LCM) of all the periods in the task set. The basic idea of using hyper-period is that once we find a profitable (for energy or vulnerability) schedule for one hyper-period, the exactly same schedule can be applied to subsequent hyper-periods.

[3]It will be precise to call n as the total number of "jobs" as in real-time system terminology. However, for ease of discussion, we do not distinguish between tasks and jobs.

underlying scheduling algorithm. EDF fetches the task with the highest priority (earliest deadline) to execute. The cache configuration selection algorithm will pick a configuration for this task and try to satisfy Equation 2 and 3 if possible. Our heuristic approach chooses between the VAEO cache configuration and performance optimal (PO) cache configuration for this task.

- ***VAEO cache configuration*** of a task is the configuration which satisfies Equation 2 and consumes the least energy among all possible configurations.
- ***PO cache configuration*** of a task is the configuration which has the shortest execution time, but PO configuration might not satisfy Equation 2.

Algorithm 1: Inter-task Cache Reconfiguration

Input: List of ready tasks (LRT) and task profile table.
Output: VAEO or PO cache configuration.
Step 1: Sort all tasks in LRT by priority and fetch the task t_c with highest priority.
Step 2: t_1 to t_m are tasks left in LRT, from highest to lowest priority. τ represents the current time.
/** check the schedulability of each task in LRT **/
for $j = 1$ *to* m **do**
  if $\tau + p_{t_c}^{PO} + \sum_{i=1}^{j} p_{t_i}^{PO} > D_{t_j}$ **then**
    Discard task t_j
  end
end
Step 3: Select cache configuration for current task t_c. Let m' be the number of tasks in LRT left after Step 2.
/** test the feasibility of using VAEO config for t_c **/
if $\tau + p_{t_c}^{VAEO} > D_{t_c}$ **then**
  $OK_{VAEO} = false$;
end
else
  $OK_{VAEO} = true$;
  for $j = 1$ *to* m' **do**
    if $\tau + p_{t_c}^{VAEO} + \sum_{i=1}^{j} p_{t_i}^{PO} > D_{t_j}$ **then**
      $OK_{VAEO} = false$;
    end
  end
end
if $OK_{VAEO} == true$ **then**
  return VAEO configuration for task t_c
end
else
  return PO configuration for task t_c
end

The intuition behind our approach of choosing between PO and VAEO configuration are as follows:

(1) The VAEO configuration satisfies the vulnerability constraint in Equation 2 and it is most beneficial for energy savings, although it might have long execution time. We would like to always choose the VAEO configuration for energy optimization, as long as this choice would not cause the current task or any of the subsequent tasks to violate their deadlines.

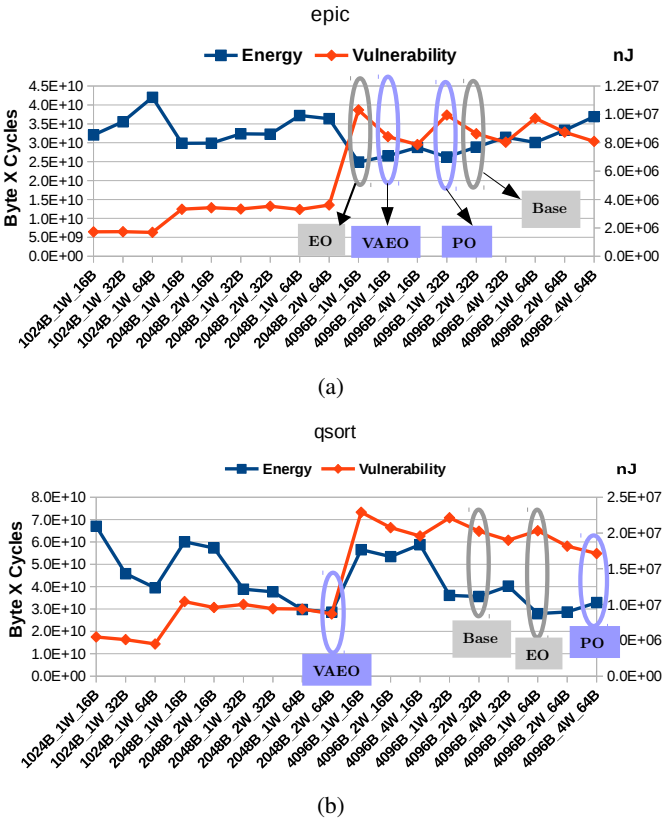


Fig. 4: VAEO and PO configurations of benchmarks *epic* and *qsort*.

(2) The PO configuration is aimed on Equation 3 for satisfying timing constraints. If the VAEO configuration of a task causes deadline violations, we would conservatively choose the PO configuration instead. With this task running under the PO configuration, the subsequent tasks will have more slack time for scheduling and possibly save energy.

Figure 4 shows the VAEO and PO configurations for benchmarks *epic* and *qsort*. The *base cache* configuration is 4096B_2W_32B. For *epic*, the PO configuration (4096B_1W_32B), determined by runtime (which is not shown in this figure), has worse vulnerability than Base. The VAEO configuration (4096B_2W_16B) has the minimum energy consumption, among all candidates which has smaller vulnerability than Base. Cache sizes of 1K and 2K (the first nine configurations) are candidates with very small vulnerability, however, they are not chosen because of large energy consumption. The configuration with minimum energy consumption (4096B_1W_16B, marked as EO in Figure 4a) is not chosen as VAEO, because its vulnerability is higher than the Base. For *qsort*, the VAEO configuration (2048B_2W_64B) finds a sweet spot which has low vulnerability and energy footprint. It is not the one with the minimum energy consumption (EO), while it has much lower vulnerability than Base and PO configurations. VAEO configuration has cache size 2K and line size 64B, which indicates that the data of the program can fit into 2K cache and data is accessed in large chunks.

Algorithm 1 illustrates the runtime cache selection procedure for VAEO approach. Let us assume that our system uses

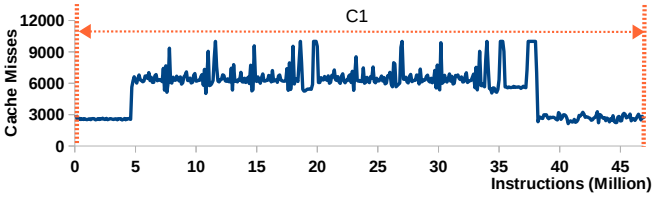
non-preemptive EDF scheduling for the task set. Tasks arrive periodically and currently available tasks will be put into the list of ready tasks (LRT), which is maintained as a priority queue based on the deadlines of tasks. Algorithm 1 is called when the processor is ready to execute a new task. The term $p_{t_i}^{PO}$ stands for the execution time of task t_i using its PO configuration, and $p_{t_i}^{VAEO}$ stands for the execution time using its VAEO configuration.

Step 1 fetches the current task t_c to be executed, which is the highest priority task from LRT. Step 2 checks the schedulability of the tasks left in LRT, when the current task t_c is executed with PO cache configuration. The schedulability of each task t_j left in the LRT is checked by $\tau + p_{t_c}^{PO} + \sum_{i=1}^{j} p_{t_i}^{PO} > D_{t_j}$, which tests whether its deadline can be met with the assumption that all preceding tasks (and itself) use PO cache configurations. If t_j cannot satisfy its deadline even with this conservative assumption, t_j should be discarded. The discarding process is done from highest priority to lowest priority, so as to achieve fewest discarded tasks. This step ensures that all tasks in LRT will satisfy their deadlines with their PO configurations, when the current task t_c is executed with its PO configuration. This step will be skipped if LRT is empty. In Step 3, we try to test the feasibility of using its VAEO configuration for the current task t_c, which will help improve vulnerability and energy consumption. The appropriate cache configuration for the current task t_c is selected by checking whether it is safe to use its VAEO configuration. VAEO configuration is safe, only if no tasks in the LRT will fail to meet their deadlines with their PO configurations. If the VAEO configuration is not safe for t_c, we will conservatively execute the current task t_c with its PO configuration, which can ensure all tasks left in the LRT to satisfy their deadlines with their PO configurations (otherwise they would have already been discarded in Step 2). This algorithm has time complexity of $O(m \log m)$ where m is the total number of tasks in LRT, since Step 1 takes $O(m \log m)$ time, Step 2 takes $O(m)$ time and Step 3 takes $O(1)$ time.

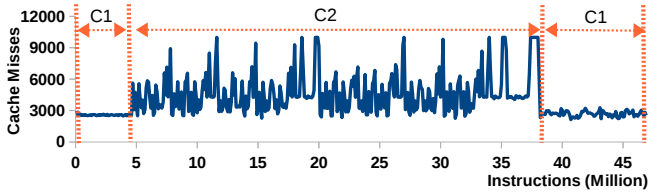
V. Intra-task Cache Reconfiguration

The heuristic approach described in Section IV provides solution for the inter-task cache reconfiguration of a task set, which will choose between the PO and VAEO configurations for each task. In this section we find that it is even more beneficial if we can reconfigure inside the task itself (i.e. intra-task reconfiguration). In other words, both intra-task and inter-task can be used simultaneously for improving vulnerability and energy efficiency. A task can have considerably different behaviour depending on which portion of execution is examined.

Figure 5(a) shows the data cache misses, when benchmark *qsort* is executed with a fixed cache C1 (1024B_1W_32B). We can observe three program phases based on the number of cache misses per sampling point (100K instructions). The first and third phases ($0 \sim 5$ million and $38 \sim 47$ million) have fewer than 3000 misses per sampling point, while the second phase has more than 6000 misses per sampling point. Based on this observation, a large cache C2 (4096B_2W_32B)



(a) One phase for the whole task using a fixed cache.



(b) Three intra-task phases using different caches

Fig. 5: Data cache misses for benchmark *qsort*. (a) Without intra-task reconfiguration, using one cache for the whole task; (b) With intra-task reconfiguration, using a different cache for each of the three phases.

is selected for the second phase in Figure 5(b), which greatly reduces the cache misses. This example shows that intra-task reconfiguration can reduce cache misses. Our ultimate goal of using intra-task reconfiguration is to further optimize energy and/or vulnerability of the task.

A phase of task (program) can be defined as an interval of execution during which a measured program metric is relatively stable. Intra-task cache reconfiguration aims at finding the sequence of cache assignments to different phases of the task, so that cache misses (energy and/or vulnerability) of this task is further optimized. In order to improve energy consumption and vulnerability, we need to properly define the phases and carefully select the configuration for each phase. Our approach for intra-task reconfiguration is to switch to the most beneficial cache configuration when the task enters a different phase during its execution. At the beginning of a new phase, we choose a configuration based on the characteristics (cache requirement) of the new phase. The cache will be flushed if we decide that the configuration is to be changed for the new phase. The flushing of cache will result in additional cache misses, which cause penalty in performance and energy consumption. However, the flushing of cache is beneficial to reduce vulnerability, because vulnerable data in cache are prematurely written back to the memory. A beneficial configuration would be one which can save energy and reduce vulnerability, in spite of the additional misses at the very beginning of the new phase.

The problem of intra-task cache reconfiguration boils down to solving the following two problems: (1) how to monitor the execution of the task and define phase partitions; (2) how to decide the cache configuration for each phase. The following sections address these challenges.

A. Phase Extraction

We introduce our approach for phase extraction in Algorithm 2. We get the cache miss statistics of all intervals for a

fixed cache (1024B_1W_32B). We have done experiments to choose different cache configurations to inspect the fluctuation of cache misses during the execution of programs. We find that the configuration (1024B_1W_32B) can reflect the program behaviors more effectively compared with larger cache sizes for our benchmarks.

Algorithm2 works in two steps: (1) identify potential phase boundaries, and (2) post-process to select profitable phases. Firstly, an interval is marked as a potential starting point of a new phase if the change in number of cache misses exceeds the threshold that we have set (line 5-7). Secondly, each of the potential phases is examined and the ones which mark a relatively stable execution (i.e. longer than the minimum threshold) are kept (line 10-12). A phase will be merged with the previous phase if it lasts no longer than $PhaseLength$. For each benchmark, it is divided into 100 sampling intervals with equal number of instructions. Each interval is profiled with the number of cache misses by simulation using a configuration of 1024B_1W_32B. We used the threshold for change of cache misses ($MissFactor$) as 2, and the threshold for minimum phase length ($PhaseLength$) as 5 intervals.

Algorithm 2: Phase Identification

```
1  Input: Benchmark, threshold MissFactor (changes of
     cache misses), threshold PhaseLength
2  Output: Identified phases.
3  Get the cache misses (CM_i) statistics of all intervals
4  for each interval i do
5      if CM_i < CM_{i-1}/MissFactor or
         CM_i > CM_{i-1} * MissFactor then
6          Set interval i as a potential phase boundary
7      end
8  end
9  for each potential phase f_j do
10     if length(f_j) < PhaseLength then
11         Merge this phase with previous phase
12     end
13 end
```

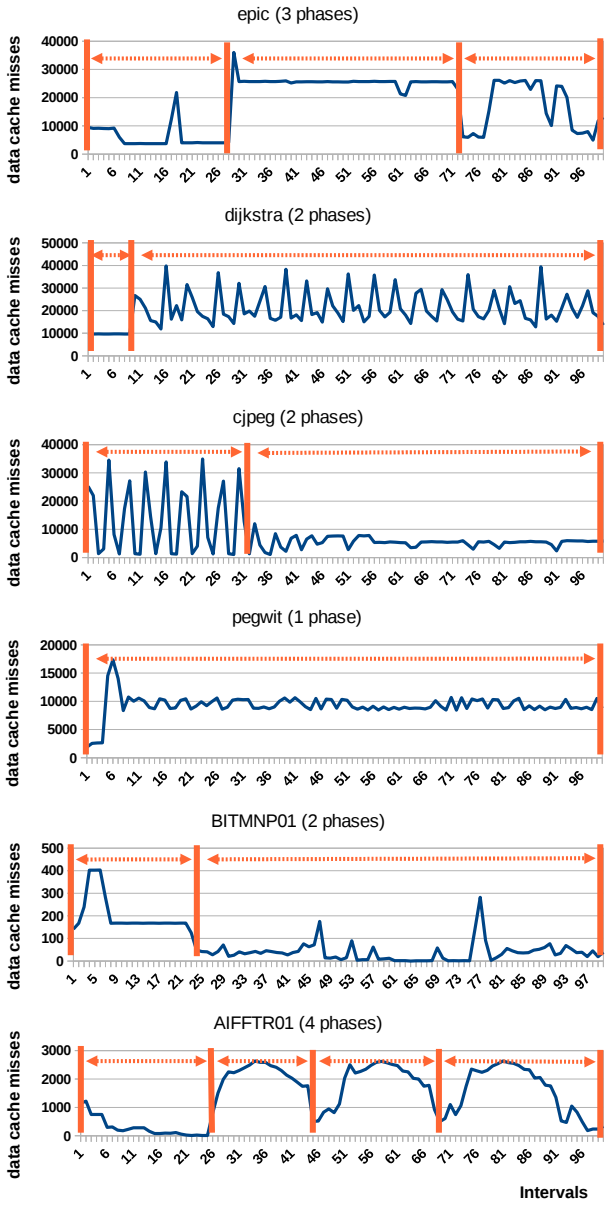


Fig. 6: Phases identified for different benchmarks.

Figure 6 shows the phases identified for 6 benchmarks from the MediaBench [22] and EEMBC [23] automotive benchmark suits. We can observe that benchmarks have very different patterns of data cache misses during execution. We identify 3 phases in epic, 2 phases in dijkstra, 2 phases in cjpeg, 2 phases in BITMNP01, and 4 phases in AIFFTR01, while pegwit has no obvious phases.

B. Cache Assignment for Phases

In this section, we show the cache assignment for phases of a task for the VAEO problem. We want to minimize the total energy consumption of all phases, with the total vulnerability constrained. The best VAEO cache configuration for m phases (f_1 to f_m) can be defined as:

$$\text{minimize} \sum_{i=1}^{m} e_{f_i}^{c_j} \tag{4}$$

$$\text{subject to} \sum_{i=1}^{m} v_{f_i}^{c_j} \leq V \tag{5}$$

$e_{f_i}^{c_j}$ and $v_{f_i}^{c_j}$ are the energy consumption and vulnerability of phase f_i using config c_j. V is the threshold for vulnerability, which is the vulnerability of the task when it is executed with the base cache.

For each of the m phases, there are n possible configurations. The time complexity for a brute-force exploration of all possible combinations is $O(n^m)$. We observe that this is essentially a dynamic programming problem, where each phase is dependent on the previous phase. If the current phase

chooses a different configuration from the previous phase, the cache needs to be flushed before running the current phase. However, if the chosen configuration for the current and previous phases are the same, the cache is not flushed and will keep the data. We define the *dynamic programming* problem as follows.

$$E^{c_j}_{(f_1 \sim f_i)} = \min_{k \in (1..n)} \{E^{c_k}_{(f_1 \sim f_{i-1})} + e^{c_j}_{f_i}\} \quad (6)$$
$$\text{where } i > 1,\ 1 \le j \le n,$$
$$\text{and } V^{c_k}_{(f_1 \sim f_{i-1})} + v^{c_j}_{f_i} \le V_{(f_1 \sim f_i)}$$

$$\text{with the initial states:} \quad E^{c_j}_{(f_1 \sim f_1)} = \begin{cases} e^{c_j}_{f_1}, & v^{c_j}_{f_1} \le V_{f_1} \\ \infty, & \text{otherwise} \end{cases} \quad (7)$$

$E^{c_j}_{(f_1 \sim f_i)}$ is the minimum total energy consumption for the first i phases ($f_1 \sim f_i$), assuming that the current phase f_i chooses c_j. Eq. (6) shows the formula to get the current minimum energy consumption, based on the previous iteration step. $V_{(f_1 \sim f_i)}$ is the threshold for vulnerability of the first i phases, which is the vulnerability when the task is run with base cache. Eq. (7) shows the initial states for our dynamic programming.

Algorithm 3 is an iterative implementation of our cache assignment approach for the phases. We use two arrays to store the energy and vulnerability values ($E[m][n]$ and $V[m][n]$), where m is the number of phases, and n is the number of cache configurations. In line 4-10, we initialize the states of phase f_1, as directed by Eq. (7). For each configuration c_j, its energy value will be updated only if its vulnerability is smaller than V_{f_1}. In line 12-25, we evaluate the states of phase f_2 to f_m, as outlined by Eq. (6). In each iteration (phase f_i), we update the optimal energy value ($E[i][j]$, which is $E^{c_j}_{(f_1 \sim f_i)}$ in Eq. (6)) for each configuration (c_j). This is done in line 15-23, which compares all solutions ($E[i-1][k]$ and $V[i-1][k]$) found at the previous iteration for phases $f_1 \sim f_{i-1}$. In the process of comparing previous solutions (line 18), we also ensure that the vulnerability constraints are not violated (line 17). In line 27-30, we iterate through feasible solutions at the last phase f_m, and find the optimal solution with the minimum energy. The initialization process of line 4-10 is of complexity of $O(n)$. The dynamic programming process of line 12-25 has complexity of $O(mn^2)$. The final iteration for output of line 27-30 has complexity of $O(n)$. Thus, the algorithm has an overall time complexity of $O(mn^2)$. This algorithm can be completed in reasonable time since m is typically less than 10 and n is 18 in our framework.

Algorithm 3: Cache Assignment for Intra-task Phases

```
1  Initialize the energy array E[m][n] = {∞, ..., ∞}
2  Initialize the vulnerability array V[m][n] = {∞, ..., ∞}
3  /** Phase 1 **/
4  for config c_j=c_1 to c_n do
5      Get e^{c_j}_{f_1} and v^{c_j}_{f_1} by running phase f_1 with config c_j
6      if v^{c_j}_{f_1} ≤ V_{f_1} then
7          E[1][j] = e^{c_j}_{f_1}
8          V[1][j] = v^{c_j}_{f_1}
9      end
10 end
11 /** Phase 2 to Phase m **/
12 for phase f_i=f_2 to f_m do
13     for config c_j=c_1 to c_n do
14         E[i][j] = ∞
15         for config c_k=c_1 to c_n do
16             Get e^{c_j}_{f_i} and v^{c_j}_{f_i} by running f_i with config c_j
17             if V[i-1][k] + v^{c_j}_{f_i} ≤ V_{f_i} then
18                 if E[i-1][k] + e^{c_j}_{f_i} < E[i][j] then
19                     E[i][j] = E[i-1][k] + e^{c_j}_{f_i}
20                     V[i][j] = V[i-1][k] + v^{c_j}_{f_i}
21                 end
22             end
23         end
24     end
25 end
26 /** Find the optimal solution with minimum energy **/
27 for config c_j=c_1 to c_n do
28     E_min = min(E[m][j], E_min);
29     V = V[m][j];
30 end
31 The path leading to E_min is the VAEO solution
32 return the VAEO config vector for all phases
```

C. Inter+Intra Cache Reconfiguration

Up to this point, we have introduced both inter-task DCR and intra-task DCR. The inter-task DCR approach optimizes at the task level, where each task is deemed as an atom since our system is non-preemptive. The intra-task DCR approach optimizes at the phase level (inside a task), where phases can execute with the intra-task VAEO configuration vector (one configuration for each phase). It is straightforward to introduce our (inter+intra)-task DCR approach, which combines these two levels of optimization by applying intra-task DCR on each task for inter-task DCR. Algorithm 4 shows our (inter+intra)-task DCR approach. In Step 1, we generate the profile (i.e., the intra-task VAEO configuration vector) for each task, which can be obtained by the phase identification and cache assignment methods described earlier in this section. Step 2 shows the (inter+intra)-task cache reconfiguration approach. We fetch the task with the highest priority from the task queue as the current task t_c. The inter-task reconfiguration method (Algorithm 1) is called to make the decision whether the intra-task VAEO configuration is suitable for t_c to satisfy deadline constraints. If the intra-task VAEO configuration is chosen, the system will execute the task with intra-task reconfiguration. Otherwise, the system will execute the task with PO configuration without intra-task reconfiguration.

Instead of using a fixed VAEO configuration for a task, our (inter+intra)-task DCR approach can use the intra-task VAEO configuration, which has optimal configurations for different phases. There is no context switching or preemption during the execution of a task, even though intra-task optimization is applied. Compared with inter-task DCR, the inter+intra DCR approach introduces overhead in the form of cache flushing

Algorithm 4: (Inter+intra)-task Cache Reconfiguration

Step 1: Generate profile for each task.
for *each task* t_i **do**
    // Call Algo. 2
    $phases = PhaseIdentify(t_i)$
    // Call Algo. 3
    Intra-task VAEO config = $CacheAssign(phases)$
end
Step 2: (Inter+intra)-task Cache Reconfiguration
while *task queue is not empty* **do**
    Fetch the current task t_c with highest priority
    Use Algo. 1 for inter-task cache reconfiguration
    if *Intra-task VAEO config is chosen* **then**
        Execute t_c with intra-task reconfiguration
    end
    else
        Execute t_c with the PO config
    end
end

when cache configurations change between phases. Since the number of phases in a task is relatively small, the overhead caused by intra-task reconfiguration is negligible (less than 1% penalty for performance). The overhead of cache flushing on energy consumption and vulnerability is far outweighed by the benefits of intra-task reconfiguration, which will be presented in our experiments.

VI. Experiments

A. Experimental Setup

The configurable caches used in our work are from the cache architecture introduced in [4]. The underlying cache architecture contains a configurable cache with a four-bank cache with sizes of 1 KB, 2 KB and 4 KB, line sizes of 16 bytes, 32 bytes and 64 bytes, and associativity of 1-way, 2-way and 4-way. In order to quantify reliability-aware DCR trade-off, we selected benchmarks from MediaBench [22] and EEMBC automotive [23] benchmark suites. Table I shows our four task sets with three selected benchmarks in each set. All of the tasks are executed with the default input parameters provided with the benchmark suites. The benchmarks from MediaBench have about 10∼200 million instructions, while the benchmarks from EEMBC AutoBench have about 1∼10 million instructions. The rationale for us to form a task set is that the tasks are of comparable size in terms of number of instructions. Both task set 1 and set 2 consist of three tasks from MediaBench. Both task set 3 and set 4 consist of three tasks from EEMBC AutoBench. Thus, set 1 and set 2 have more instructions and can potentially stress the cache with more cache accesses, compared to set 3 and set 4.

	Task 1	**Task 2**	**Task 3**
Task Set 1	epic	dijkstra	cjpeg
Task Set 2	fft	pegwit	qsort
Task Set 3	AIFFTR01	AIFIRF01	BITMNP01
Task Set 4	CACHEB01	CANRDR01	IIRFLT01

TABLE I: Four task sets with twelve benchmarks

We modified the SimpleScalar simulator [24] for cache vulnerability analysis and energy consumption estimation. We performed the vulnerability analysis during cache accesses for each byte in instruction and data cache. The vulnerability estimation function collects all the vulnerable intervals for each valid byte in cache. We applied the same energy model as in [4] to calculate both dynamic and static energy consumption, and the energy consumption was estimated using CACTI 5.3 [25] with 65 nm technology. For static profiling of each task to find the PO, VAEO, and Intra-task VAEO (with intra-task reconfiguration) cache configurations, we developed Perl scripts to search the design space of all possible cache configurations. Since we only consider systems with one level of reconfigurable cache architecture, the space of possible cache configurations is small. The statistics for all possible cache configurations for a task can be collected in a reasonable time (a few hours). Once we have the profile tables for all the tasks, we use an EDF scheduler to simulate the system for a hyper-period. The cache selection algorithms are integrated in the scheduler to make decisions to reconfigure the cache during simulation. The optimization for instruction cache and data cache are independent. In the following subsections, we will first present results for optimization of individual benchmarks, followed by the results for task sets with scheduling and cache selection.

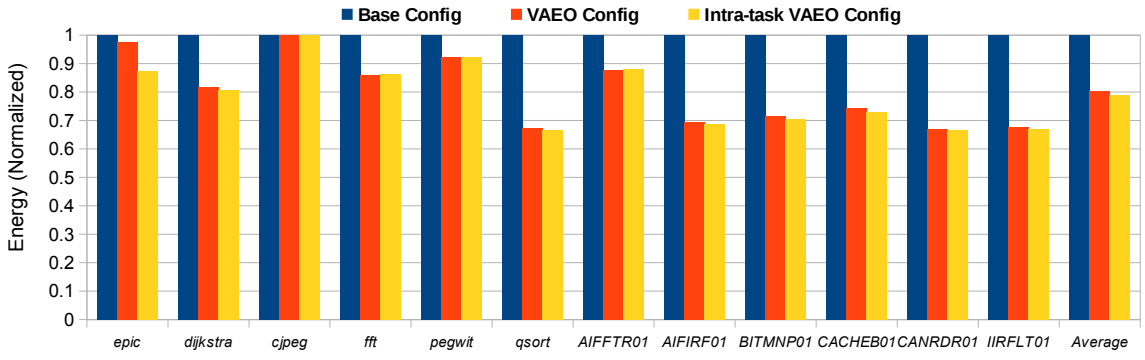
B. VAEO and Intra-task VAEO configurations of single tasks

In this section, we present the results to show the effectiveness of reconfiguration for single tasks. We will compare the energy consumption and vulnerability when a task is executed with the Base, VAEO and Intra-task VAEO configurations.

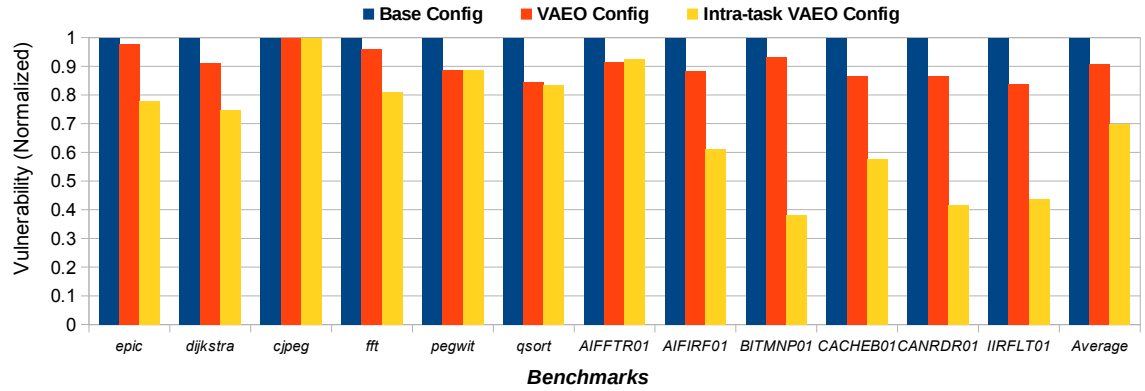
- **Base Config**: the configuration of the *base cache*, which is 4KB, 2-way associative with line size of 32 bytes.
- **VAEO Config**: the vulnerability-aware energy optimal configuration without intra-task reconfiguration.
- **Intra-task VAEO Config**: the VAEO configuration when intra-task reconfiguration is allowed.

Figure 7a and 7b show the energy and vulnerability of L1 data cache for 12 benchmarks. The VAEO configurations can reduce energy consumption (up to 33.0%, 19.8% on average), as well as vulnerability (up to 16.1%, 9.3% on average), compared with Base configurations. The Intra-task VAEO configurations can reduce energy consumption (up to 33.5%, 21.1% on average), as well as vulnerability (up to 58.4%, 30.0% on average), compared with Base configurations. Generally speaking, the VAEO configurations can greatly reduce energy and vulnerability compared with the Base, and the Intra-task VAEO can further improve energy and vulnerability compared with VAEO.

We observe that the trend of energy and vulnerability improvement in data cache (Figure 7) is similar to the trend of instruction cache (Figure 8). Figure 8a and 8b show the energy and vulnerability of L1 instruction cache for 12 benchmarks with their Base, VAEO and Intra-task VAEO configurations. The VAEO configurations can reduce energy consumption (up to 34.3%, 20.4% on average), as well as vulnerability (up to 47.6%, 25.3% on average), compared with Base configurations. The Intra-task VAEO configurations can reduce energy



(a) Data Cache Energy Consumption



(b) Data Cache Vulnerability

Fig. 7: Comparison of DL1 energy consumption and vulnerability of single tasks for Base, VAEO and Intra-task VAEO configurations.

consumption (up to 34.5%, 23.1% on average), as well as vulnerability (up to 68.9%, 29.5% on average), compared with Base configurations.

Compared with Figure 7 for data cache, the intra-task VAEO configs for instruction cache (Figure 8) tend to have very similar energy and vulnerability numbers as the inter-task VAEO configs. There are two reasons for this: (1) Compared with data cache, instruction cache is more sensitive to reconfiguration. It means that the beneficial configuration points are not that many. For example, if we change the instruction cache size from 4KB to 2KB, it significantly impacts execution time, which makes huge impact on energy consumption and vulnerability. (2) Reconfiguration in the middle of execution (intra-task reconfiguration) will flush the instruction cache first. Starting a new phase with an empty instruction cache will tend to significantly impact performance. This effect will propagate and impact energy and vulnearbility. Thus, the selected intra-task VAEO configurations tend to be very similar to the selected VAEO configurations.

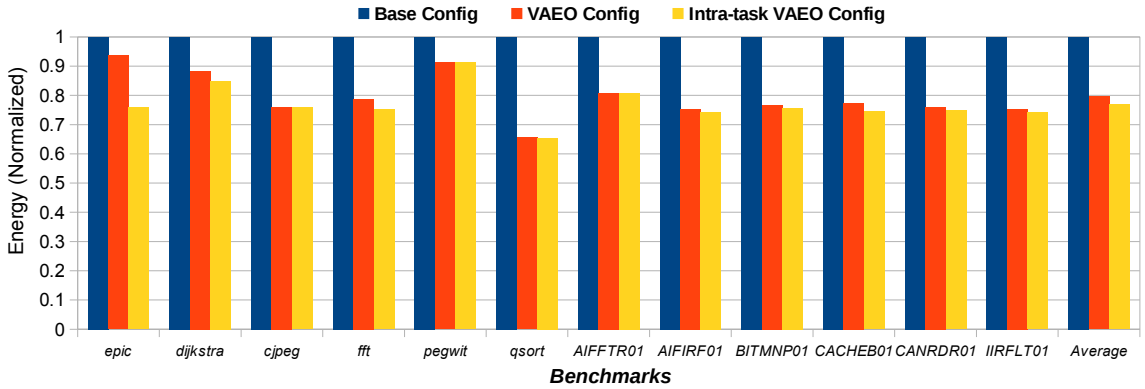
For benchmark $pegwit$ and $cjpeg$, the energy and vulnerability numbers for *VAEO Config* and *Intra-task VAEO Config* are exactly the same. This is because *Intra-task VAEO Config* is exactly the same as *VAEO Config*. (1) For $pegwit$, we identify the whole program as one phase (as seen in Figure 6), because the miss rate remains almost the same. (2) For $cjpeg$, we identify two phases but the reconfiguration algorithm chooses the same configuration for the two phases. In other words, we didn't do intra-task reconfiguration for these two benchmarks, thus the results remain exactly the same for Figure 7 and 8.

C. Results for Inter-task VAEO and (Inter+Intra)-task VAEO

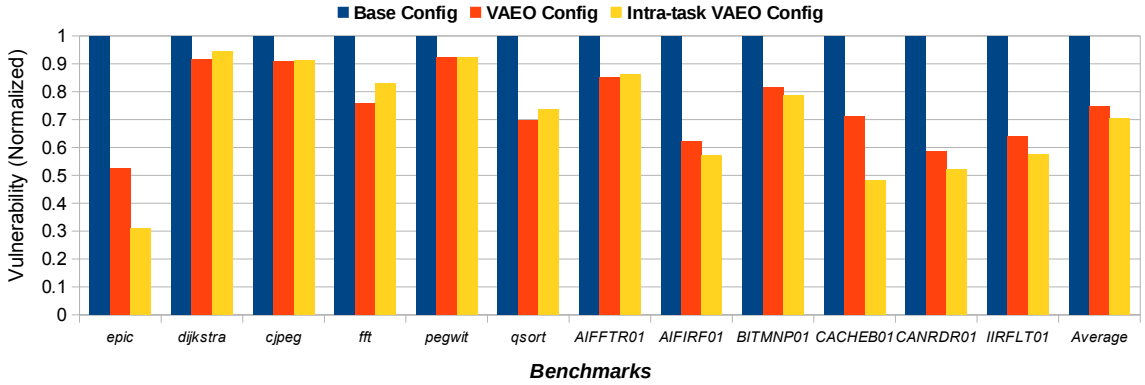
In this section, we present the results to show the effectiveness of reconfiguration for task sets using proposed approaches. We profile each task with its PO, EO, VAEO, and Intra-task VAEO configurations. The runtime algorithms (Algorithm 1 for inter-task reconfiguration and Algorithm 4 for (inter+intra)-task reconfiguration) will select between PO and VAEO (or Intra-task VAEO) configurations. In the following section, we compare our proposed VAEO approaches with the base cache system as well as [7]:

- **Base** refers to the base system which uses the fixed *Base Config* for all tasks.
- **EO [7]** refers to the Energy-Optimization approach in [7] which chooses between PO and EO configurations.
- **Inter-task VAEO** is our inter-task reconfiguration approach when the runtime algorithm chooses between PO and VAEO configurations.
- **(Inter+Intra)-task VAEO** is our (inter+intra)-task reconfiguration approach when the runtime algorithm chooses between PO and Intra-task VAEO configurations.

Figure 9a and 9b show the results of L1 data cache for four task sets. *Inter-task VAEO* can improve energy by 19.6% on average and vulnerability by 9.0% on average, compared



(a) Instruction Cache Energy Consumption



(b) Instruction Cache Vulnerability

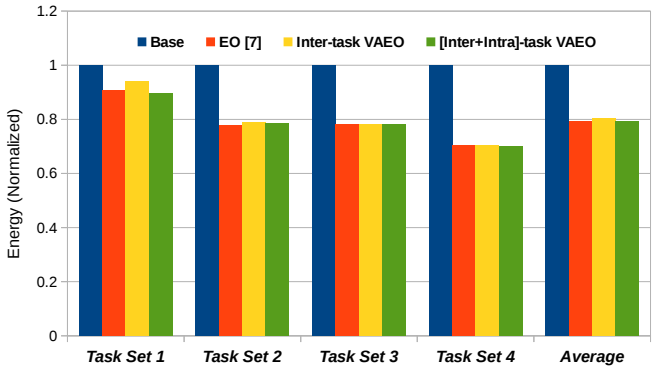
Fig. 8: Comparison of IL1 energy consumption and vulnerability of single tasks for Base, VAEO and Intra-task VAEO configurations.

with *Base*. *(Inter+Intra)-task VAEO* can improve energy by 20.9% on average and vulnerability by 25.3% on average, compared with *Base*. *(Inter+Intra)-task VAEO*, which takes advantage of both intra-task and inter-task reconfiguration, can further improve energy consumption compared with *Inter-task VAEO*. Compared with EO [7], our VAEO approach can reduce vulnerability by 8.7% on average, while it consumes 1.2% more energy on average. This minor energy penalty is not surprising since EO did not consider any vulnerability threshold during energy minimization, whereas our approach respects the vulnerability constraint. Our (inter+inter)-task VAEO approach reduce vulnerability by 24.9% and it saves 0.1% more energy compared with EO [7]. As shown in Figure 9b, EO [7] produces very bad vulnerability for Task Set 1 and Task Set 2, which is even worse than the *Base* system.

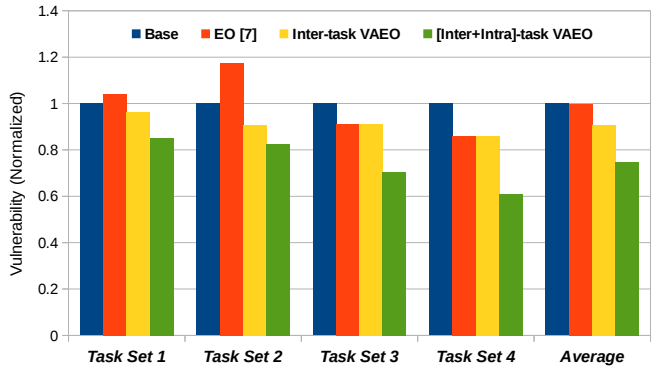
Figure 10a and 10b show the results of L1 instruction cache for four task sets. *Inter-task VAEO* can improve energy by 21.6% on average and vulnerability by 24.1% on average. *(Inter+Intra)-task VAEO* can improve energy by 23.8% on average and vulnerability by 28.2% on average. Compared with EO [7], our VAEO approach produces the exact same results for four task sets. This is because the VAEO configurations are exactly the same as EO configurations for IL1 cache. This suggests that IL1 cache accesses has similar patterns among benchmarks, thus the results for IL1 have fewer variations than that of DL1 cache. Our (inter+inter)-task VAEO approach can improve a little bit further over the VAEO approach. Compared with EO [7], our (inter+inter)-task VAEO approach can reduce vulnerability by 4.1% and save 2.3% more energy.

D. Hardware Overhead

Cost of implementation involves two factors: (i) the cost of reconfiguration infrastructure; (ii) the cost of chip area for storing profile table. As mentioned in the Section II, dynamic cache reconfiguration (DCR) is an approach widely used in embedded systems for performance improvement and energy saving. This architecture requires very simple hardware augmentation and minor overhead [7]. The overhead to implement our VAEO approach is mostly the cost to store the profile table in hardware. The cache tuner will fetch the cache configuration information from the profile table. The size of the table depends on the number of tasks in the system and the information needed to store for each task. For the VAEO approach, we need to store two configurations (i.e., [*Config*, *Runtime*] for the PO and VAEO configurations) for each task. Five bits are used to specify a configuration since the configurable cache architecture used in this study offers 18 possible cache configurations. Another 16 bits are used to store the expected runtime of the task. For 12 benchmarks, the profile table contains 24 entries each with 21 bits. For the VAEO approach with intra-task reconfiguration, we need to store [$Phase_1$, $Config_1$, ..., $Phase_n$, $Config_n$, *Runtime*] for the intra-task VAEO configuration. We use 16 bits ($Phase_i$) to indicate the

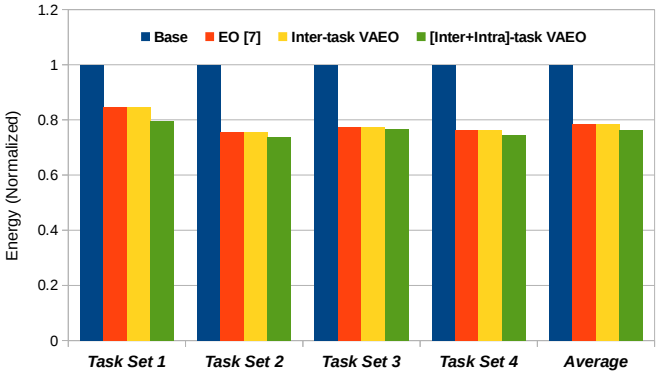


(a) Data Cache Energy Consumption

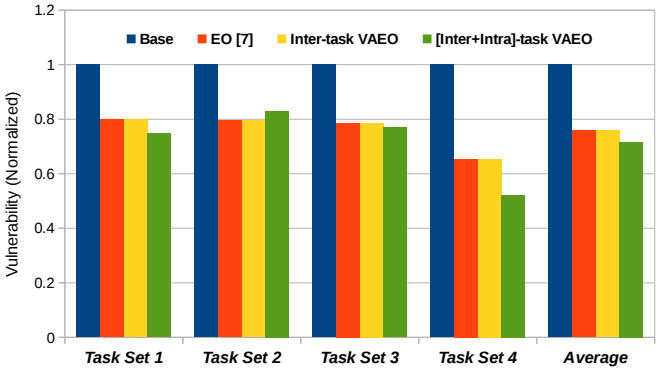


(b) Data Cache Vulnerability

Fig. 9: Data cache energy and vulnerability.



(a) Instruction Cache Energy Consumption



(b) Instruction Cache Vulnerability

Fig. 10: Instruction cache energy and vulnerability.

start instruction number of the phase, and 5 bits to store its cache configuration. For benchmarks used in our paper, n is at most 4. In total it takes at most 100 (=16*4+5*4+16) bits to store the intra-task VAEO configuration for one task.

TABLE II: Overhead of Profile Table (65nm technology)

Approach	Table size (bits)	Area (μm^2)	Dynamic Power (μW)	Leakage Power (μW)
VAEO	504	10641	19.26	243.37
Intra-task VAEO	1032	21788	39.44	498.33

We used Synopsis Design Compiler with TSMC library to implement the profile table. We estimate a table lookup frequency of once per 3 μs. It is a table lookup every one thousand instructions using a 500 MHz CPU with an average CPI of 1.5. It should be suffice since the benchmarks we used have around 1 to 200 million instructions. Table II shows our results of the area, dynamic power, and leakage power for the profile table using 65nm technology. We observed that on average for each task set, the energy overhead of our approach accounts for less than 2% (0.067 mJ compared to 3.38 mJ) of the total energy savings for VAEO approach, and less than 3% (0.14 mJ compared to 4.73 mJ) of the total energy savings for intra-task VAEO approach. The (intra+intra)-task VAEO approach has slightly higher overhead than VAEO and also has higher energy savings. Therefore, we concludes that the overhead of profile tables is negligible compared to the energy savings for both VAEO and (inter+intra)-task VAEO approaches.

VII. Conclusions

Dynamic cache reconfiguration is widely used for improving energy and performance in embedded systems. While cache vulnerability is a well studied area, previous research efforts did not explore cache vulnerability in the context of cache reconfiguration. In this paper, we developed algorithms to reduce cache vulnerability with energy and performance considerations. Our experimental results demonstrated that our approach can significantly improve the reliability of both instruction and data caches. For the data cache, the Inter+Intra DCR approach can improve energy by 20.9% on average and vulnerability by 25.3% on average. For the instruction cache, the Inter+Intra DCR approach can improve energy by 23.8% on average and vulnerability by 28.2% on average. Future work will focus on applying our approach to more flexible systems in broader areas. (1) Our approach can be extended to multi-level caches in single-core systems as well as multicore systems. The only difference would be that heuristic approaches should be used to efficiently explore beneficial cache configurations because exhaustive exploration may not be feasible since the number of possible configurations can be very large. (2) Our approach can be extended to systems allowing preemptive execution. This can be achieved by partitioning tasks into phases and profiling each partition, and preemptive task can resume execution using the configuration for the current phase.

REFERENCES

[1] Y. Huang and P. Mishra,"Reliability and energy-aware cache reconfiguration for embedded systems," In Proceedings of the International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, 2016, pp. 313-318.

[2] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 76). IEEE Computer Society Press, 2012.

[3] R. Jeyapaul and A. Shrivastava, "Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors," In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), Taipei, 2011, pp. 105-114.

[4] W. Wang, S. Ranka, P. Mishra, "Dynamic Reconfiguration in Real-Time Systems - Energy, Performance, Reliability and Thermal Perspectives," Springer, 2012.

[5] C. Ekelin, "Clairvoyant non-preemptive EDF scheduling," In Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06), Dresden, 2006, pp. 7.

[6] X. Qin, W. Wang and P. Mishra, "TCEC: Temperature- and Energy-Constrained Scheduling in Real-Time Multitasking Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 31(8), pages 1159-1168, August 2012.

[7] W. Wang, P. Mishra and A. Gordon-Ross, "Dynamic Cache Reconfiguration for Soft Real-Time Systems," ACM Transactions on Embedded Computing Systems (TECS), volume 11, issue 2, Article 28, 31 pages, July 2012.

[8] W. Wang and P. Mishra, "System-Wide Leakage-Aware Energy Minimization using Dynamic Voltage Scaling and Cache Reconfiguration in Multitasking Systems," IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI), 20(5), pages 902 - 910, 2012.

[9] W. Wang, P. Mishra and S. Ranka, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multi-Core Systems", In Proceedings of the ACM/IEEE Design Automation Conference (DAC), pp. 948-953, 2011.

[10] Y. Huang and P. Mishra, "Vulnerability-Aware Energy Optimization Using Reconfigurable Caches in Multicore Systems," 2017 IEEE International Conference on Computer Design (ICCD), Boston, MA, 2017, pp. 241-248.

[11] Y. Cai, M. Schmitz, A. Ejlali, B. Al-Hashimi and S. Reddy, "Cache size selection for performance, energy and reliability of time-constrained systems," In Proceedings of Asia and South Pacific Conference on Design Automation (ASP-DAC), 2006, pp. 6.

[12] S. Mittal and J. S. Vetter, "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems," in IEEE Transactions on Parallel and Distributed Systems, 27(4), pp. 1226-1238, 2016.

[13] Y. Lyu and P. Mishra, A Survey of Side Channel Attacks on Caches and Countermeasures, Springer Journal of Hardware and Systems Security (HASS), 2(1), pages 33-50, 2018.

[14] Y. Ko, R. Jeyapaul, Y. Kim, K. Lee and A. Shrivastava, "Guidelines to design parity protected write-back L1 data cache," In Proceedings of the ACM/IEEE Design Automation Conference (DAC), 2015, pp. 1-6.

[15] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor." In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2003, p. 29-.

[16] W. Zhang et al. "An analysis of microarchitecture vulnerability to soft errors on simultaneous multithreaded architectures." IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS) 2007.

[17] V. Sridharan et al. Memory Errors in Modern Systems The Good, The Bad, and The Ugly. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2015.

[18] J. Suh et al. Soft error benchmarking of L2 caches with PARMA. ACM SIGMETRICS Performance Evaluation Review 2011.

[19] V. Sridharan, H. Asadi, M. B. Tahoori and D. Kaeli, "Reducing Data Cache Susceptibility to Soft Errors," in IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 4, pp. 353-364, Oct.-Dec. 2006.

[20] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli, "Balancing Performance and Reliability in the Memory Hierarchy," In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005 (ISPASS), pp. 269-279.

[21] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," In Proceedings of the 32nd annual International Symposium on Computer Architecture, 2005 (ISCA), pp. 532-543.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communication systems," In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30). 1997, pp. 330-335.

[23] http://www.eembc.org. EEMBC, The Embedded Microprocessor Benchmark Consortium.

[24] http://www.simplescalar.com. The SimpleScalar Simulator.

[25] http://www.hpl.hp.com/research/cacti/. CACTI.

[26] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," In Proceedings of the 30th annual International Symposium on Computer Architecture (ISCA). ACM, 2003, pp. 336-349.

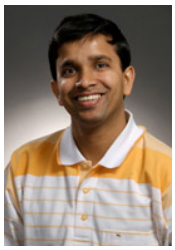
[27] H. Hajimiri and P. Mishra, "Intra-Task Dynamic Cache Reconfiguration," In Proceedings of the 25th International Conference on VLSI Design, 2012, pp. 430-435.

[28] T. Adegbija, A. Gordon-Ross and A. Munir, "Dynamic phase-based tuning for embedded systems using phase distance mapping," In Proceedings of the IEEE 30th International Conference on Computer Design (ICCD), 2012, pp. 284-290.

[29] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache for low energy embedded systems," ACM Trans. Embed. Comput. Syst. 4, 2 (May 2005), pp. 363-387.

[30] A. Shrivastava, J. Lee, and R. Jeyapaul. "Cache vulnerability equations for protecting data in embedded processor caches from soft errors," In Proceedings of the ACM conference on Languages, compilers, and tools for embedded systems (LCTES '10), pp. 143-152, 2010.

Yuanwen Huang received the B.E. degree from Huazhong University of Science and Technology, China, in 2012. He received the Ph.D. degree in Computer Engineering from University of Florida, in 2017. He was a recipient of the Best Paper Award from the International Symposium on Quality Electronic Design in 2016. His research interests include energy and reliability optimization in embedded systems, hardware security and trust for integrated circuits.



Prabhat Mishra (S00-M04-SM08) is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include embedded and cyber-physical systems, energy-aware computing, hardware security and trust, system-on-chip verification, bioinformatics, and post-silicon debug. He received his Ph.D. in Computer Science and Engineering from the University of California, Irvine. He has published six books and more than 150 research articles in premier international journals and conferences. His research has been recognized by several awards including the NSF CAREER Award, IBM Faculty Award, three best paper awards, and EDAA Outstanding Dissertation Award. Prof. Mishra currently serves as the Deputy Editor-in-Chief of IET Computers & Digital Techniques, and as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Transactions on VLSI Systems, and Journal of Electronic Testing. Prof. Mishra is an ACM Distinguished Scientist and a Senior Member of IEEE.