Cache Reconfiguration Using Machine Learning for Vulnerability-aware Energy Optimization

ALIF AHMED, YUANWEN HUANG, and PRABHAT MISHRA, University of Florida

Dynamic cache reconfiguration has been widely explored for energy optimization and performance improvement for single-core systems. Cache partitioning techniques are introduced for the shared cache in multicore systems to alleviate inter-core interference. While these techniques focus only on performance and energy, they ignore vulnerability due to soft errors. In this article, we present a static profiling based algorithm to enable vulnerability-aware energy-optimization for real-time multicore systems. Our approach can efficiently search the space of cache configurations and partitioning schemes for energy optimization while task deadlines and vulnerability constraints are satisfied. A machine learning technique has been employed to minimize the static profiling time without sacrificing the accuracy of results. Our experimental results demonstrate that our approach can achieve 19.2% average energy savings compared with the base configuration, while drastically reducing the vulnerability (49.3% on average) compared to state-of-the-art techniques. Furthermore, the machine learning technique enabled more than 10x speedup in static profiling time with a negligible prediction error of 3%.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Reliability**; *Real-time systems*; • **Hardware** → Power and energy;

Additional Key Words and Phrases: Vulnerability, dynamic cache reconfiguration, cache partitioning, machine learning, energy optimization, multicore

ACM Reference format:
Alif Ahmed, Yuanwen Huang, and Prabhat Mishra. 2019. Cache Reconfiguration Using Machine Learning for Vulnerability-aware Energy Optimization. *ACM Trans. Embed. Comput. Syst.* 18, 2, Article 15 (March 2019), 24 pages.
https://doi.org/10.1145/3309762

1 INTRODUCTION

Multicore architectures consist of multiple processor cores to improve execution performance of application programs. A multicore processor usually has on-chip caches to resolve the performance bottleneck caused by the increasing gap between processor and memory speed. In a typical multicore system, each core maintains its private L1 caches while all cores share the same L2 cache. There are many optimization techniques for multi-level on-chip caches to improve

This work was partially supported by the National Science Foundation under grant CNS-1526687. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

Authors' addresses: A. Ahmed, 137 Yellowstone Dr. Apt 106, Charlottesville, VA-22903; email: alifahmed@ufl.edu; Y. Huang and P. Mishra, 432 Newell Drive, Gainesville, FL-32611; emails: {yuanwenhuang, prabhat}@ufl.edu.

1539-9087/2019/03-ART15 $15.00
https://doi.org/10.1145/3309762

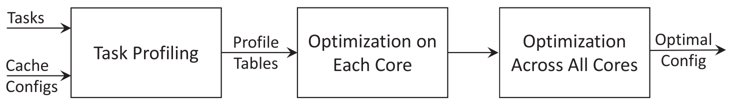

Fig. 1. Overview of the three-step optimization process.

performance and energy consumption of the overall system [11, 15, 24, 27, 28, 30, 42, 46, 48]. With the increasing demand for high reliability and availability, vulnerability of caches due to soft errors is gaining increasing importance. Data corruption caused by soft errors can change the behavior of applications and may eventually result in a system failure. As for performance and energy improvement, it is beneficial to maintain a useful data longer in the cache. However, longer data retention can negatively impact the vulnerability or probability of data corruption due to soft errors. It is a great challenge to keep vulnerability under control while we optimize the cache subsystem for improvement in performance and energy consumption.

Application-based techniques on cache optimization have been very effective in improving performance and energy consumption. One of the most successful techniques for cache energy optimization is dynamic cache reconfiguration (DCR). The basic idea of DCR is to select a suitable cache configuration to satisfy the specific data access behavior of the application. By tuning the cache configuration (cache size, associativity, and line size) at runtime, it is possible to optimize the energy consumption and improve performance of different applications. DCR has been well studied for energy savings in both uniprocessor [47] and multicore systems [48]. Recent work by Huang et al. [18] studies the impact of DCR on vulnerability in the L1 caches for a uniprocessor. However, there are no existing efforts in vulnerability-aware cache reconfiguration for multicore systems.

As for a shared L2 cache, it may cause performance degradation because of data contentions among different cores. Cache partitioning (CP) techniques are introduced to alleviate this problem by judiciously dividing the shared cache and mapping a dedicated partition of the cache to each core. CP can improve the performance of independent tasks running on different cores, by eliminating inter-task interference on the shared cache. DCR and CP are both cache optimization techniques to properly tune the cache subsystem based on the data access pattern of applications. Previous work by Wang et al. [48] explores the idea of combining DCR and CP for energy optimization in real-time multicore systems. However, their work does not consider vulnerability.

In this article, we propose a vulnerability-aware energy optimization technique that integrates cache reconfiguration (DCR) of private L1 caches and cache partitioning (CP) of the shared L2 cache. It is composed of three steps as depicted in Figure 1. A profile table is built for each task during the task profiling step. This profile table contains energy, runtime, and vulnerability measurements for all configurations. A small number of table entries is used to train a model, which then predicts the rest of the table. The second step finds the best L1 configuration on each core for all L2 configurations using dynamic programming. The third step optimizes across cores by finding the best L2 partition scheme. We get the optimal L1 and L2 configuration by the end of the three-step process. Note that getting the optimal configuration is dependent on profiling the task-sets. This is not a limiting factor for embedded systems, where, often times, the designers have prior knowledge on what applications will run on the system.

Our article makes the following important contributions:

— We explore the inter-dependence of L1 DCR and L2 CP for performance and energy consumption, as well as vulnerability.
— Our proposed algorithm is able to minimize energy consumption without violating both vulnerability and real-time constraints.

— Our fast and scalable static profiling algorithm can efficiently search the design space of L1 configurations and L2 partitions, making it feasible to find the optimal result using dynamic programming.
— We proposed a machine learning based technique to further reduce the static profiling time by more than 10 times, utilizing highly accurate models.
— Our results demonstrate that our approach can provide significant energy savings compared with the base configuration as well as drastic reduction in vulnerability compared to the state-of-the-art techniques.

The remainder of the article is organized as follows. Related approaches are surveyed in Section 2. The architecture model and a motivational example are presented in Section 3. Section 4 discussed our approach for vulnerability-aware optimization in detail. Section 5 presented the experimental results, and Section 6 concludes the article.

2 BACKGROUND AND RELATED WORKS

2.1 Cache Vulnerability Mitigation and Measurement

In this article, we used the term *vulnerability* to denote cache vulnerability caused by soft errors. When a charged particle hits a transistor, it will create a depletion region. A lot of electrons get attracted to the depletion region. Because of this, being hit by a charged particle might disconnect a conducting drain and source. If used as cache, such phenomena may lead to bit-flip of the stored value from 1 to 0 or from 0 to 1. Caches occupy a majority of the chip area, and are made of tightly packed transistors—making them highly vulnerable to soft errors [25]. Studies have shown that soft errors can cause a random data failure every 3 to 30 days for a 100 megabit memory [8, 29]. To mitigate this issue, we can either prevent soft errors, or detect and correct errors after it happens [22, 35, 36]. Several techniques such as early write-back and periodic flashing are proposed in Ref. [3]. These techniques work by keeping an updated copy of the cached data in the main memory and recovering data in case of a soft error. Write-through cache can also be used for the same purpose. However, such cache designs increase bus usage which in turn negatively impacts performance. Parity checking, checksum, cyclic redundancy check (CRC), and error-correcting codes (ECC) are alternatives to this approach [3, 32]. However, these approaches may not be suitable for applications with short access time constraints [3]. In this article, we made no assumption on the error detection or correction techniques for caches. Our approach determines optimum configuration to minimize energy and vulnerability, independent of error detection and correction methodology. For critical applications, our technique can be used to minimize soft errors, and then ECC can be applied on top of it to recover from the escaped errors.

Vulnerability is quantitatively measured by splitting a bit's lifetime into vulnerable and unvulnerable intervals [5, 17, 18, 26, 37, 51]. The bit is vulnerable when occurrence of a soft error will make the program to get corrupted data. On the other hand, effect of soft error can get masked by some activity. In such cases, a program will not get corrupted data, and the bit is in an in vulnerable state. Figure 2 shows an example [17]. The vulnerable time is indicated by a bold black line. We can have read, write, or eviction operations during the lifetime of a bit in the cache. Suppose we write a value X to a bit. Now, if a soft error changes the bit value to X', then the next read by the program will get the corrupted value of X'. So the write-read interval is a vulnerable interval. Similarly, a read followed by another read is also a vulnerable interval, as the reads can get different values because of the soft error. The bit is also vulnerable from a write to the eviction as can be seen from Figure 2(a). A soft error during this interval can cause the corrupted data to be written to the main memory. On the other hand, read-write and write-write intervals are not vulnerable. This is because the last write will mask the effect of soft error. The read-evict interval in Figure 2(b) is not

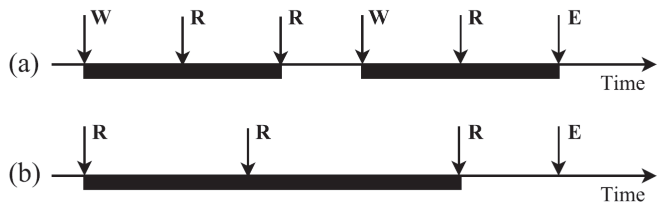

Fig. 2. Vulnerable intervals of two data elements in a cache without soft-error protection (where W=Write Access, R=Read Access, E=Evict). (a) Data with both write and read accesses; (b) data with only read accesses [17].

vulnerable since the data will not be written back to main memory. Cache vulnerability is simply measured by summing up the vulnerable intervals for all bytes [17, 18, 20].

2.2 Dynamic Cache Reconfiguration and Partitioning

Reconfigurable cache architectures are extensively studied in Refs [12, 16, 38–40, 43–45, 49, 50]. Gordon-Ross et al. [9] utilized DCR to improve performance by online feedback and dynamic self-tuning of the cache. An energy-efficient approach using DCR is proposed in Ref. [47] for soft real-time systems by using static profiling and dynamic reconfiguration. DCR in two-level cache hierarchy in uniprocessor is studied in Ref. [41]. DCR for multicore systems is studied in Ref. [15] for thread-fairness and performance improvement. Wang et al. proposed an energy-efficient approach for multicore systems in Ref. [48] by using DCR on private L1 caches and CP on the shared L2 cache. Authors in Refs [2], [13], and [34] have further divided the tasks into smaller phases. Cache usage is almost same within a phase, but varies greatly between phases. DCR is applied on individual phases instead of tasks.

CP is a special case of reconfiguration on the shared cache among multiple cores [24, 33]. Initial works of CP aim at improving the performance of multicore systems [21, 33]. Liang et al. explored cache partitioning on GPUs at task-level granularity [23]. Reddy et al. investigates energy-efficient CP for multitasking embedded systems in Ref. [31]. Chakraborty et al. proposed working set size based cache resizing to reduce energy consumption [7]. However, none of the above approaches takes vulnerability into consideration.

Cai et al. [6] is the first to consider cache configuration (only cache size selection) for energy and vulnerability in time-constrained systems. Huang et al. [18] proposes a DCR approach for performance, energy, and vulnerability tradeoffs in uniprocessor-based systems. To the best of our knowledge, the proposed work is the first attempt in studying vulnerability-aware optimizations in multicore systems in the presence of reconfigurable caches.

3 MODELING SYSTEMS WITH RECONFIGURABLE CACHES

In this section, we describe the modeling of multicore systems with reconfigurable caches. First, we describe the underlying multicore architecture. Next, we present the energy and vulnerability models. Then, we provide an illustrative example to motivate the need for the proposed exploration framework. Finally, we present the problem formulation.

3.1 Multicore Architecture Model

Figure 3 shows a typical multicore system with a shared on-chip L2 cache and private L1 caches for each core. In this article, we assume that the private L1 caches (both IL1 and DL1) are reconfigurable, and the shared L2 cache is equipped with way-based partitioning. The L1 caches can reconfigure its cache size, associativity, and line size. The reconfigurable cache architecture

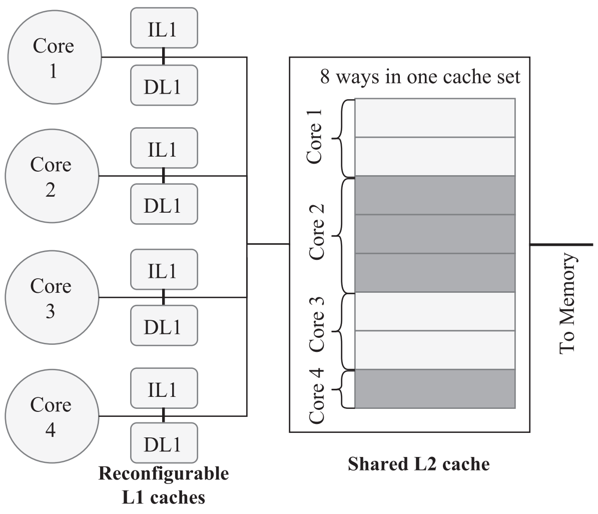

Fig. 3. A multicore system with reconfigurable L1 caches and a partition-enabled shared L2 cache.

is the same as Refs [9] and [47]. The cache size is tuned by selectively shutting down the banks with gated-V_{dd} techniques. The associativity is reconfigured by logically concatenating ways. The line size can be changed by fetching multiple unit-length blocks in one access. The reconfigurable architecture is lightweight, which introduces negligible overhead [47].

The shared L2 cache with way-based partitioning [33] is illustrated in Figure 3. Each L2 cache set (8-way associativity as in this example) is partitioned into four parts, each of which will be assigned to one core. Each core will access only the assigned portion of the cache sets and enforce the LRU replacement policy among its individual group of ways. The number of ways assigned to a core is referred to as its *partition factor*. As shown in Figure 3, Core 1 has an L2 partition factor of 2. In this article, we use dynamic reconfiguration of the L1 caches and static partitioning of the shared L2 cache. In other words, L1 cache configurations can be tuned for each application on each core during runtime. While L2 partition factors are pre-determined for each core and remain unchanged during runtime, all applications running on that core have the same L2 partition factor.

3.2 Energy and Vulnerability Models

The *Energy Model* is adopted from the one used in Ref. [47]. The cache energy consumption consists of static and dynamic energy: $E = E_{sta} + E_{dyn}$. The static energy dissipation E_{sta} is computed as $E_{sta} = P_{sta} \times t$, where P_{sta} is the static power of cache. Dynamic energy dissipation E_{dyn} comes from both cache accesses and cache misses.

$$E_{dyn} = Accesses \times E_{access} + Misses \times E_{miss} \tag{1}$$

$$E_{miss} = E_{offchip\_access} + E_{block\_fill}, \tag{2}$$

where E_{access} and E_{miss} are the energy required per cache access and per cache miss, respectively. E_{access} and E_{miss} are constant values for one specific configuration. $E_{offchip\_access}$ is the energy for accessing the lower level of the memory hierarchy, and E_{block_fill} is the energy for filling the cache block with fetched data.

The *Vulnerability Model* is based on per-byte analysis of the cached data as discussed in Section 2.1. Similar to Ref. [18], the lifetime of a byte is divided into vulnerable and un-vulnerable intervals. The measured vulnerability V is the summation of vulnerable intervals as shown in

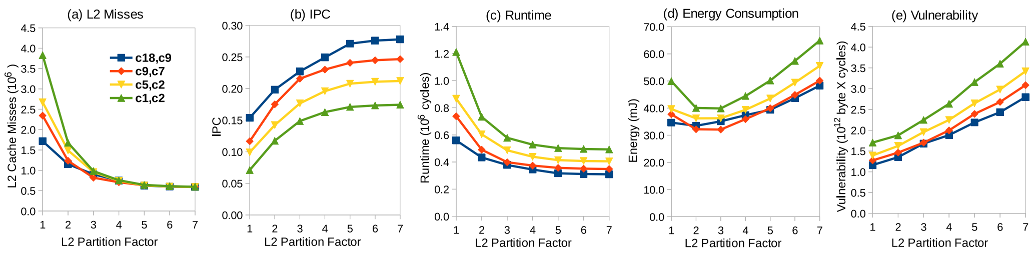

Fig. 4. Inter-dependence of L1 DCR and L2 CP on (a) L2 Misses, (b) IPC, (c) Runtime, (d) Energy, and (e) Vulnerability.

Equation (3) [18].

$$V = \sum_{all\ bytes} vulnerable\ interval\ of\ byte \tag{3}$$

3.3 Illustrative Example

Figure 4 shows the impact of L1 DCR and L2 CP for benchmark *qsort* from MiBench [10]. The L2 partition factor can change from 1 to 7 in an 8-way associative L2 cache. Four pairs of cache configurations[1] for IL1 and DL1 are randomly chosen. We observe that different L1 configurations will lead to different L2 cache misses (Figure 4(a)) and pipeline throughput (i.e., IPC in Figure 4(b)). This is expected since L1 configurations determine the number accesses to the L2 cache, as well as the pipeline throughput. Secondly, as L2 partition factor w increases, L2 cache misses will decrease and eventually converge (for $w \geq 4$) for different L1 configurations. However, the IPC shows great diversity even when the L2 partition factor is large.

Figure 4(c)–(e) show the runtime, energy consumption, and cache vulnerability of the benchmark, respectively. It is interesting to see that they have different patterns as L2 partition factor w increases. Runtime will decrease drastically as w increases, which is accordant with the pattern of IPC. Energy consumption will decrease to a minimal point (for $w = 3$), but it will increase when w becomes larger. This is because dynamic energy (caused by a lot of cache misses) dominates the total energy consumption when w is small, while static energy dominates when w is too large. However, vulnerability will increase with w. This is expected for two reasons: (1) a large w means that L2 cache has more valid area and is holding more data, which remains vulnerable to soft errors; (2) the decrease in cache misses (data replacement) also indicates that data are residing in the cache for a longer time, which means data will have longer vulnerable intervals. While a large L2 partition facilitates performance, it might jeopardize energy consumption and vulnerability. This shows that performance, energy, and vulnerability have very different (often conflicting) cache requirements.

Given the above observations, both L1 DCR and L2 CP have a major impact on performance, energy consumption, and vulnerability. The interesting tradeoffs between them is the motivation of this article to explore for optimization. We exploit L1 DCR and L2 CP simultaneously for vulnerability-aware energy optimization for real-time multi-core systems.

3.4 Problem Formulation

We model our multicore system as follows:

— The multicore processor has m cores $\mathbb{P}\ \{p_1, p_2, \ldots, p_m\}$.
— Each core has private IL1 and DL1, both of which can be reconfigured to r configurations $\mathbb{C}\ \{c_1, c_2, \ldots, c_r\}$.

[1]Here, $c18$ and $c9$, for example, stand for the IL1 and Dl1 using the 18^{th} and 9^{th} configuration, respectively.

- The shared L2 cache is ω-way associative, which supports way-based partitioning.
- A set of n independent tasks $\mathbb{T}$ $\{\tau_1, \tau_2, \ldots, \tau_n\}$ with a common deadline D.

Our optimization goal is to find a reconfiguration scheme **R** for the private L1 caches and a partitioning scheme **P** for the shared L2 cache such that the overall energy consumption E is minimized without violating vulnerability constraints and task deadlines. Assume that we are given the following:

- A task mapping scheme **M**: $\mathbb{T} \rightarrow \mathbb{P}$, which assigns tasks to each core. In this article, we assume that the task mapper **M** is given, which can ensure that the total runtime on each core is comparable. ρ_k is the number of tasks mapped to core k.
- A reconfiguration scheme **R** for L1 caches: $C_I, C_D \rightarrow \mathbb{T}$, which assigns one IL1 and DL1 configuration to each task.
- A partitioning scheme **P** for L2 cache: $\mathbf{P} = \{w_1, w_2, \ldots, w_m\}$, which allocates w_k ways to core k.

For task $\tau_{k,i} \in \mathbb{T}$ (the i_{th} task on core k), $e_{k,i}(c_I, c_D, w_k)$ denotes the energy consumption of the cache subsystem when the task is executed with L1 configurations (c_I, c_D) and L2 partition factor w_k. Similarly, let $t_{k,i}(c_I, c_D, w_k)$ and $v_{k,i}(c_I, c_D, w_k)$ denote the execution time and the total vulnerability, respectively. Our minimization problem can be formulated as follows:

$$E = \sum_{k=1}^{m} \sum_{i=1}^{\rho_k} e_{k,i}(c_I, c_D, w_k) \tag{4}$$

is minimized subject to:

$$\max_{k=1..m} \left(\sum_{i=1}^{\rho_k} t_{k,i}(c_I, c_D, w_k) \right) \leq D \tag{5}$$

$$\sum_{i=1}^{\rho_k} v_{k,i}(c_I, c_D, w_k) \leq V_k, \forall k \in [1, m] \tag{6}$$

$$\sum_{k=1}^{m} w_k = \omega; w_k \geq 1, \forall k \in [1, m] \tag{7}$$

Equation (5) guarantees that all tasks will meet the deadline D. Equation (6) guarantees that the total vulnerability of the tasks on each core is constrained by the threshold V_k, which is chosen as the base case vulnerability. Equation (7) verifies that the partitioning scheme is valid.

4 VULNERABILITY-AWARE DCR+CP

In this section, we present our approach, which utilizes the static profiles of tasks to efficiently search the design space for the optimal energy solution. Our three-step optimization approach is illustrated in Figure 5: (i) the **first step** profiles each task for all possible configurations; (ii) the **second step** uses a dynamic programming algorithm to determine the optimal L1 cache configurations on each core; (iii) the **third step** combines the solutions from step two by evaluating all feasible L2 partition schemes. The remainder of this section describes these steps in detail.

4.1 Task Profiling

Theoretically, we can do static profiling for all possible L1 reconfiguration schemes **R** and all possible L2 partition schemes **P** with all possible task set combinations in $\mathbb{T}$. However, this exhaustive

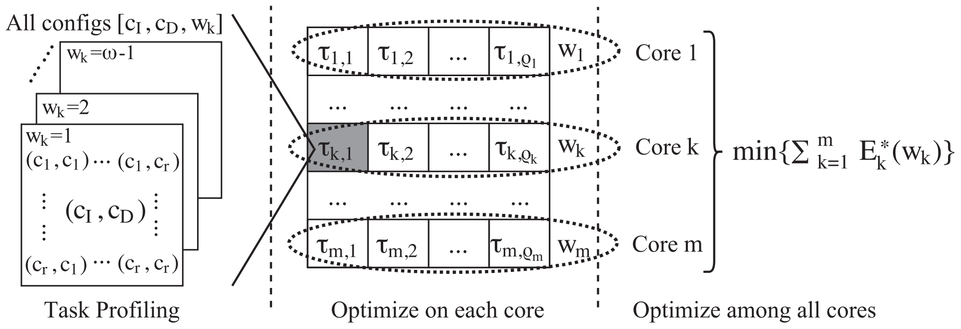
All configs [c_I, c_D, w_k]
w_k=ω-1
w_k=2
w_k=1
(c_1, c_1) ··· (c_1, c_r)
(c_I, c_D)
(c_r, c_1) ··· (c_r, c_r)
τ_1,1 τ_1,2 ... τ_1,ϱ1 w_1
τ_k,1 τ_k,2 ... τ_k,ϱk w_k
τ_m,1 τ_m,2 ... τ_m,ϱm w_m
Core 1
Core k
Core m
min{Σ_{k=1}^m E_k^*(w_k)}
Task Profiling
Optimize on each core
Optimize among all cores

Fig. 5. Three-step optimization: the first step statically profiles each task, the second step optimizes for each partition factor on each core to find the best L1 cache configurations, and the third step combines the optimal solution on all cores to find the best L2 partition scheme.

approach requires excessive simulation time. Let's say we have to profile n tasks, the number of IL1/DL1 configurations is r, and the number of L2 partition factors is ω. Then the total number of simulations with the exhaustive approach will be:

$$\sum_{k=1}^{n} \binom{n}{k} \cdot r^2 \cdot (\omega - 1)$$

Here, r^2 is the number of IL1 and DL1 combinations and $(\omega - 1)$ is the number of possible L2 partition factors. Fortunately, the profiling complexity can be drastically reduced by exploiting the independence of tasks in the system [19]. We can profile each task as if it is executed independently on a uniprocessor with a w_i-way associative L2 cache (with capacity equal to w_i/ω of the original L2). That means we need to simulate each task with all possible IL1 and DL1 cache configurations, along with all possible L2 partition factors to build up their respective profile table. Therefore, each task has a profile table with $r^2 \cdot (\omega - 1)$ entries, each of which contains the runtime, energy consumption, vulnerability for the corresponding L1 configurations, and L2 partition factor. Evaluating each entry requires a simulation. Thus, the total number of simulations required to fill up profile tables for n tasks would be $r^2 \cdot (\omega - 1) \cdot n$. This is a massive improvement over the exhaustive method, but not good enough. Consider an example system with 20 tasks, 18 IL1 and DL1 cache configurations, and an 8-way set-associative L2 cache [47]. This system requires $18^2 \times 7 \times 20 = 45{,}360$ simulations for building profile tables. If each simulation takes 1 minute, it will translate to 31.5 days of continuous simulation. Huang et al. [19] used this method for task profiling. In their experiments, the static profiling finished within three days by utilizing parallel execution on multiple cores. It is evident that the time requirement would be infeasible for practical applications and on systems with a large number of cache configurations. This article introduces a machine learning based technique to enable faster construction of the profile table. The basic idea is to:

- Partially build the profile tables by running a small number of simulations.
- Train a model using these entries.
- Predict the rest of the table using the trained model.

If we only use simulations as in Ref. [19], the profile table building time is the time needed to run $r^2 \cdot (\omega - 1) \cdot n$ simulations (t_{sim_all}). In case of the proposed machine learning based technique, total profile table building time consists of three components: (a) simulation time for collecting training data (t_{sim_train}), (b) model training time (t_{train}), and (c) prediction time (t_{pred}). In our

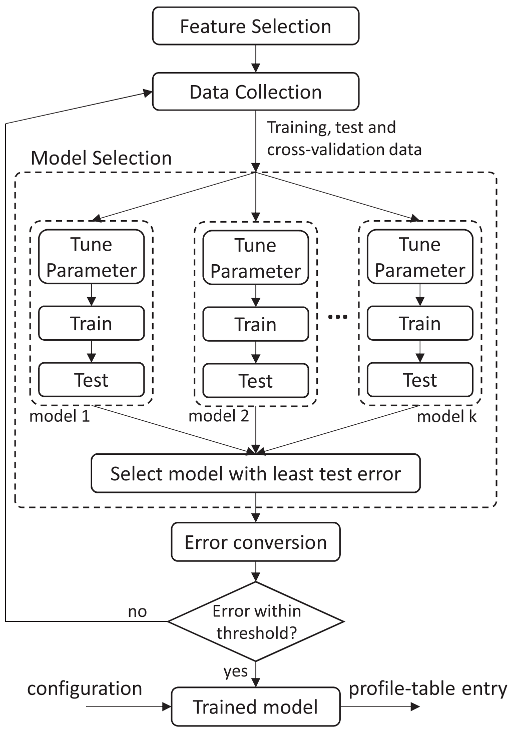

Fig. 6. Overview of profile-table construction of a task using machine learning.

experiments, we found that $t_{sim\_train} \gg t_{train} \gg t_{pred}$. Consequently, the approximate speedup of the proposed method compared to Ref. [19] is $t_{sim\_all}/t_{sim\_train}$. For example, if we simulate the tasks on 50% of the configurations (fill up 50% of the profile table, and use it for building the prediction model), we will get a speedup of approximately two times. If we simulate 5%, speedup will be nearly 20 times. However, lowering the amount of training data usually corresponds to a higher prediction error. The objective is to attain maximum accuracy while keeping the simulation time t_{sim_train} within the allowed time budget. More time budget allows for collecting more training data, resulting in less prediction error. Figure 10 captures the training data requirement trend as a function of obtained accuracy, which can help to provide an educated guess of the initial training time. As we are trying to predict energy, runtime, and vulnerability values, this is a regression problem.

Figure 6 gives an overview of task profiling using machine learning, while Algorithm 1 provides the implementation details. First, we fill up a portion of the profile table entries by simulating a task on different configurations. Next, we train several models with the collected data and select the model with the least error. If the error is within a selected threshold, then we predict the rest of the profile table entries using that model. If not, then we collect more training data by running more simulations, and repeat the procedure until the error threshold criteria is satisfied. The same process is repeated for all tasks. Note that we are building three models per task—one for predicting energy, one for runtime, and one for vulnerability. Details of these steps are given next.

Feature Selection: Selecting proper features is the most important part of a machine learning problem. As we are predicting profile table entries with different cache configurations, these configurations themselves are selected as the features. Let $X =< x_1, x_2, \ldots, x_7 >$ be the feature vector where:

ALGORITHM 1: Profile Table Generation

```
/* Model building                                              */
normalize($X_{all}$)
foreach task do
    $X_{pred} = X_{all}$
    $X_{sim} = \emptyset$
    $min\_error = \infty$
    while $X_{pred}! = \emptyset$ and $min\_error > \epsilon$ do
        $X_{sel} = randomSample(X_{pred})$
        $X_{pred} = X_{pred} - X_{sel}$
        $X_{sim} = X_{sim} + X_{sel}$
        $Y_{sel} = simulate(X_{sel})$
        $[X_{train}, X_{cv}, X_{test}] = distribute(X_{sel})$
        $[Y_{train}, Y_{cv}, Y_{test}] = distribute(Y_{sel})$
        foreach regression algorithm do
            /* Parameter sweep                                     */
            foreach param do
                $model = train(X_{train}, Y_{train}, X_{cv}, Y_{cv}, param)$
                $\hat{Y}_{test} = predict(X_{test}, model)$
                $error = nrmse(Y_{test}, \hat{Y}_{test})$
                if $error < min\_error$ then
                    $min\_error = error$
                    $sel\_model = model$
    /* Profile table entry prediction                              */
    $\hat{Y}_{pred} = predict(X_{pred}, sel\_model)$
    $Y_{all} = Y_{sim} + \hat{Y}_{pred}$
```

— x_1 : IL1 cache size
— x_2 : IL1 cache line size
— x_3 : IL1 cache associativity
— x_4 : DL1 cache size
— x_5 : DL1 cache line size
— x_6 : DL1 cache associativity
— x_7 : L2 partition factor

Here, the assumption is that we are configuring using L1 cache's size, line size, and associativity, and L2 cache's partition factor. If some other parameters are used for cache configuration, they should be included into the feature vector. Note that $c_I =< x_1, x_2, x_3 >$, $c_D =< x_4, x_5, x_6 >$, and $w_k =< x_7 >$. For each task, these configuration parameters (features) are sufficient to uniquely identify a profile table entry. Introducing additional features will unnecessarily make the learning model prone to over-fitting.[2] In the presence of more features, the over-fitting issue can be mitigated by using a large amount of training data. However, this is not a viable option in our case as we are trying to minimize the amount of training data. Adding extra features is avoided for this reason. As feature vector X is solely composed of cache configurations, we will use the term *feature vector* and *configuration* interchangeably to denote X.

Data Collection: Profile table entries are collected in this step by simulating the task with different configurations. These entries are then used for model training purposes. For the ease of explanation, assume that $Y_{<ss>}$ corresponds to the actual (simulated) entries and $\hat{Y}_{<ss>}$ are the predicted entries for the configuration set $X_{<ss>}$. Now, let X_{all} denote the set of all possible

[2]Over-fitting occurs when a model is too specific for the training data. It can be caused by too little data or too many features. Over-fitting will result in low training but high testing error.

configurations for a task. X_{sim} is the set of configurations on which we already simulated the task and have the corresponding table entries Y_{sim}. Remaining configurations are in X_{pred}. We need to predict the profile table entries for these configurations. It is evident that $X_{all} = X_{sim} + X_{pred}$ and $|X_{all}| = r^2 \cdot (\omega - 1)$.

Initially, all the table entries are empty. Thus, $X_{pred} = X_{all}$ and $X_{sim} = \emptyset$. In the subsequent iterations, some of the configurations from X_{pred} are selected for simulation (X_{sel}). After simulating with X_{sel} configurations, Y_{sel} entries are put into a profile table. Consequently, X_{sel} configurations are removed from X_{pred} and added to X_{sim}. Size of X_{sel} determines the number of simulations carried out in each iteration. We have fixed $|X_{sel}|$ to $0.05 * r^2 \cdot (\omega - 1)$ in our experiments. This effectively means that 5% of the total table entries are collected in each iteration. More flexible schemes are also possible. For example, $|X_{sel}|$ can be decreased in steps, or can be made proportional to the prediction error.

Model Selection: In this step, we train multiple models with the filled table entries Y_{sim} and their configurations X_{sim}. For model building purposes, these entries and configurations are divided into three groups—training (X_{train}, Y_{train}), cross-validation (X_{cv}, Y_{cv}), and testing (X_{test}, Y_{test}). This essentially means that $X_{sim} = X_{train} + X_{cv} + X_{test}$ and $Y_{sim} = Y_{train} + Y_{cv} + Y_{test}$. The training set is used for training the model. Cross-validation is used for hyper-parameter tuning, like learning rate or regularization parameter. The test set is used for determining the model with the least error. A standard split is used in our experiments: 70% for training, 15% for cross-validation, and 15% for testing. We have used shallow neural network for training our models. These models are further tuned by hyper-parameter sweeping. For shallow neural network, a suitable hyper-parameter for sweeping is the number of hidden layer nodes. If the amount of training data is very small, a lower number of hidden layer nodes gives the best accuracy. Training with more nodes in such cases will make the model to overfit the training data. As training data increases, a higher number of nodes becomes necessary for greater accuracy. Thus hyper-parameter sweeping is required to get the optimum number of nodes. After training models with different hyper-parameters, the model that offered the lowest Mean Square Error (MSE) on the test set is finally selected. MSE is defined in Equation (8). In the equation, Y holds the actual values from simulation ($= Y_{test}$) and $\hat{Y}$ holds the predicted values ($= \hat{Y}_{test}$).

$$MSE = \frac{1}{n} \sum_{i=i}^{n} (Y_i - \hat{Y}_i)^2 \tag{8}$$

Error Checking and Profile Table Completion: In the previous phase, we have chosen the model with the lowest MSE. Now we need to determine if the prediction error is within a tolerable threshold, ϵ. Unfortunately, MSE is not a suitable measurement for prediction error. While MSE is the standard cost function for regression problems, it varies greatly from task to task. Error expressed as percentage is more intuitive and consistent across tasks. For this reason, we have used Normalized Root Mean Square Error (NRMSE) as the metric for error threshold. It is also known as the coefficient of variation of RMSE and is defined as:

$$NRMSE = 100\% * \sqrt{MSE}/\bar{Y} \tag{9}$$

Here, $\bar{Y}$ is the average value. For error threshold calculation, NRMSE is measured over the test dataset, Y_{test} and $\hat{Y}_{test}$. If NRMSE is larger than the error threshold ϵ, then we collect more simulation data and repeat the model building procedure. On the other hand, if NRMSE is within ϵ, then the model is used for predicting the rest of the profile table. Formally, input to the model will be the configuration set X_{pred} and output will be the predicted energy, vulnerability and runtime values ($\hat{Y}_{pred}$). Full profile table is built by combining predicted data $\hat{Y}_{pred}$ and simulated data Y_{sim} as shown in Figure 7.

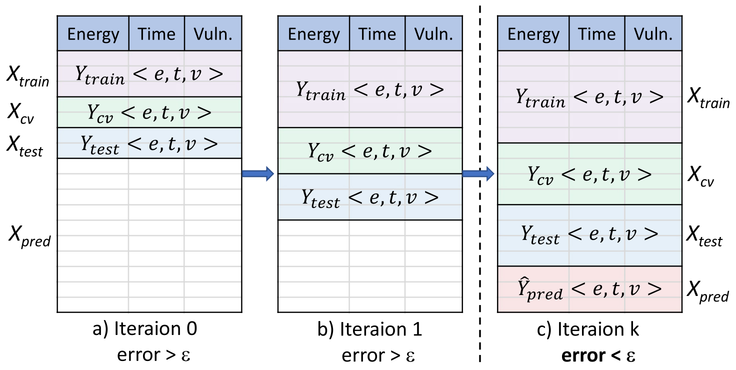

Fig. 7. Profile table generation of a task. (a) Initial iteration with error $> \epsilon$. (b) Next iteration after data collection. Error is still $> \epsilon$. (c) After k-th iteration, error becomes $< \epsilon$. Remaining table entries ($\hat{Y}_{pred}$) are predicted using this model.

One concern here is how to select appropriate error threshold. A practical approach is to start with a large error threshold (e.g., 10%), and reduce it further if the time budget permits. Simulation data that is collected during the initial runs are used for training in subsequent runs, so no simulation time is wasted. Figure 10(a) in the experimental section plots the percentage of simulation data required vs. error threshold for 20 benchmarks with different characteristics. This graph can be used to make a rough estimation of required simulation time during error threshold selection.

Note that the profiling can be done off-line. In this work, we profiled the tasks on the standard set of inputs available for those tasks. We assume that the input size remains the same but content can vary. This is a reasonable assumption for real-time embedded systems. We performed our off-line analysis by varying input patterns (data values) for all the benchmarks and observed that it has minor impact on the footprint of data access. Since profile of vulnerability and energy estimation for data pages depends on the data access pattern, our static profiling will remain effective for different input patterns. Our observations are consistent with the ones made by existing literature. Weixun et al. [47] explored the impact of input size and pattern changes, and found out that reasonable input variations have minor effect on both energy and performance optimal cache configuration.

4.2 Optimization on Each Core

In order to find the optimal solution under deadline and vulnerability constraints, we first optimize on each core (find profitable L1 configurations), and then optimize across all cores (find the best L2 partition scheme). In this section, we explain our approach for optimization on each core. Since static partitioning of L2 is used, tasks on the same core share the same L2 partition factor w_k. This fact enables us to treat each core as a subproblem, which optimizes the energy consumption for a given core under different L2 partition factors. In other words, we find cache assignment $\mathbf{R}$ to minimize $E_k(w_k) = \sum_{i=1}^{\rho_k} e_{k,i}(c_I, c_D, w_k)$ constrained by $\sum_{i=1}^{\rho_k} t_{k,i}(c_I, c_D, w_k) \leq D$ and $\sum_{i=1}^{\rho_k} v_{k,i}(c_I, c_D, w_k) \leq V_k$, with k and w_k fixed for $\forall k \in [1, m]$ and $\forall w_k \in [1, \omega - 1]$.

This subproblem is to choose L1 configurations for each task so that the total energy is optimized with constraints. The optimization goal is to minimize energy, which can be discretized to simplify the problem. We can use a dynamic programming algorithm to search for the optimal solution. Let $e_k^{min}(w_k)$ and $e_k^{max}(w_k)$ denote the minimum possible energy ($\sum_{i=1}^{\rho_k} min\{e_{k,i}(c_I, c_D, w_k)\}$) and the maximum possible energy ($\sum_{i=1}^{\rho_k} max\{e_{k,i}(c_I, c_D, w_k)\}$) on core k, respectively. The energy consumption $E_k(w_k)$ of core k using partition factor w_k is bounded by $[e_k^{min}(w_k), e_k^{max}(w_k)]$. Let S_i^E

```
If $(T[i][E] > T[i-1][E - e_{k,i}(c_I, c_D, w_k)] + t_{k,i}(c_I, c_D, w_k)$ && $V[i][E] > V[i-1][E - e_{k,i}(c_I, c_D, w_k)] + v_{k,i}(c_I, c_D, w_k))$
{
    $T[i][E] = T[i-1][E - e_{k,i}(c_I, c_D, w_k)] + t_{k,i}(c_I, c_D, w_k)$
    $V[i][E] = V[i-1][E - e_{k,i}(c_I, c_D, w_k)] + v_{k,i}(c_I, c_D, w_k)$
}
```

Fig. 8. Recursive formula for dynamic programming.

Table 1. The Time Table **T** for Dynamic Programming for Per-core Optimization

	e_k^{min}	$e_k^{min}+1$	...	e_a	e_b	...	e_c	...	e_d	...	$e_k^{max}-1$	e_k^{max}
1	t_1	∞	...	t_2	∞	...	t_3	...	t_4	...	∞	∞
2	∞	t_5	...	∞	∞	...	t_6	...	t_7	...	∞	∞
...	∞	∞	...	∞	∞	...	∞	...	∞	...	∞	∞
ρ_k	∞	∞	...	∞	∞	...	∞	...	∞	...	∞	∞

Table 2. The Vulnerability Table **V** for Dynamic Programming for Per-core Optimization

	e_k^{min}	$e_k^{min}+1$	...	e_a	e_b	...	e_c	...	e_d	...	$e_k^{max}-1$	e_k^{max}
1	v_1	∞	...	v_2	∞	...	v_3	...	v_4	...	∞	∞
2	∞	v_5	...	∞	∞	...	v_6	...	v_7	...	∞	∞
...	∞	∞	...	∞	∞	...	∞	...	∞	...	∞	∞
ρ_k	∞	∞	...	∞	∞	...	∞	...	∞	...	∞	∞

denote the current solution found for the first i tasks. It has a cumulative energy consumption of E while the execution time and vulnerability are minimized. The execution time $T[i][E]$ for S_i^E is stored in a two-dimensional table T. The vulnerability for S_i^E is stored in another two-dimensional table V. As we try out all possible (c_I, c_D) configurations, we update the solution for S_i^E whenever runtime or vulnerability can be improved. The dynamic programming process uses the recursive formula shown in Figure 8 to update the two tables. The solutions for the first i tasks (the i^{th} row in the two tables) are built upon the previous step, i.e., the $(i-1)^{th}$ row. All entries in T and V are initialized to some very large value. Based on the above recursive formula, we update the tables one row at a time for all energy values in $[e_k^{min}(w_k), e_k^{max}(w_k)]$. When the i^{th} row is calculated, all previous $(i-1)$ rows are already computed. The final optimal energy consumption $E_k^*(w_k)$ can be found by:

$$E_k^*(w_k) = \min\{E_k \mid T[\rho_k][E_k] \leq D \ \&\& \ V[\rho_k][E_k] \leq V_k\} \tag{10}$$

Equation (10) provides the solution for core k with partition factor w_k, which has minimum energy consumption with deadline and vulnerability constraints satisfied.

Tables 1 and 2 illustrate the dynamic programming bookkeeping tables for per-core optimization. Table 1 is used for the total execution time and Table 2 is used for the total vulnerability of the first i tasks assuming the total energy consumption is e. Note that we discretized the energy values so that they are numerically small to make the dynamic programming efficient. All entries in the two tables are initialized to ∞. We update the two tables simultaneously in a row-wise manner. Algorithm 2 (lines 2 to 18) iterates to find the best L1 configurations for all tasks in core k with partition factor w_k. During each iteration, all discretized energy values (e) and all L1 cache configurations (1 to r^2) for current task $\tau_{k,i}$ are examined. As we try out all possible (c_I, c_D) configurations, we update these two tables whenever runtime or vulnerability can be improved according to the recursive formula shown in Figure 8. As shown in the algorithm (lines 4 to 7), the first row of the two tables will be filled up with (t_1, v_1), (t_2, v_2), (t_3, v_3), and (t_4, v_4) using the relation in line 7. The dynamic programming process to fill up the rest of the rows are handled in lines 10 to 17. For example, we update the second rows with (t_5, v_5), (t_6, v_6), and (t_7, v_7) according to the relation in line 14. After we fill all rows (finish iterating all tasks), we get the optimal solution $E^*_k(w_k)$ from the last rows of the two tables according to line 18.

4.3 Optimization Across All Cores

In this step, we combine the solutions found on each core and search for the minimum total energy consumption E^* of all cores within all L2 partition schemes **P**. For a given partition factor w_k on core k, the optimal energy $E^*_k(w_k)$ is already calculated in the first step. A valid partition scheme $\{w_1, w_2, \ldots, w_m\}$ is one that complies with Equation (7). The final total energy E^* can be found by:

$$E^* = min\left\{\sum_{k=1}^{m} E^*_k(w_k)\right\}, \quad \forall\{w_1, w_2, \ldots, w_m\} \in \mathbf{P} \tag{11}$$

Since the number of valid partition schemes is small (35 for 4-core processor with an 8-way associative L2 cache), an exhaustive search on all partition schemes is feasible. In our experiment, we assume that after the tasks on a core finish execution, the core, along with its private L1 caches and the designated L2 partition, is turned off. Thus, E^* will be the final energy consumption for all cores running with the optimal configuration and partitioning scheme.

Algorithm 2 shows the major steps of our cache reconfiguration and partitioning approach. In the **first step** (line 1), we use Algorithm 1 to generate the profile table. In the **second step**, our algorithm iterates to find the best L1 configurations for all tasks in core k with partition factor w_k. A detailed description of this dynamic programming process is given in Section 4.2. At the end of this

ALGORITHM 2: Vulnerability-aware DCR+CP

```
   /* 1st step: Task profiling (Section 4.1)                                  */
1  profileTableGeneration() /* Algorithm 1                                    */
   /* 2nd step: Optimize on each core (Section 4.2)                           */
2  for $k = 1$ to $m$ do
3      for $w_k = 1$ to $\omega - 1$ do
4          for $e = e_k^{min}(w_k)$ to $e_k^{max}(w_k)$ do
5              for $c_I, c_D \in \mathbb{C}$ do
6                  if $e_{k,1}(c_I, c_D, w_k) == e$ then
7                      if $t_{k,1}(c_I, c_D, w_k) < T[1][e]$ && $v_{k,1}(c_I, c_D, w_k) < V[1][e]$ then
8                          $T[1][e] = t_{k,1}(c_I, c_D, w_k)$
9                          $V[1][e] = v_{k,1}(c_I, c_D, w_k)$
10         for $i = 2$ to $\rho_k$ do
11             for $e = e_k^{min}(w_k)$ to $e_k^{max}(w_k)$ do
12                 for $c_I, c_D \in \mathbb{C}$ do
13                     $e' = e - e_{k,i}(c_I, c_D, w_k)$
14                     if $T[i-1][e'] + t_{k,i}(c_I, c_D, w_k) < T[i][e]$ && $V[i-1][e'] + v_{k,i}(c_I, c_D, w_k) < V[i][e]$
15                     then
16                         $T[i][e] = T[i-1][e'] + t_{k,i}(c_I, c_D, w_k)$
17                         $V[i][e] = V[i-1][e'] + v_{k,i}(c_I, c_D, w_k)$
18         $E^*_k(w_k) = \min\{e_k \mid T[\rho_k][e_k] \le D \;\&\; V[\rho_k][e_k] \le V_k\}$
   /* 3rd step: Optimize across cores (Section 4.3)                           */
19 for all $P_j = \{w_1, w_2, \ldots, w_m\} \in \mathbf{P}$ do
20     $E^*_j = \sum_{k=1}^{m} E^*_k(w_k)$
21     $E^* = min(E^*, E^*_j)$
22 return $E^*$
```

step (line 18), we get the optimal solution $E_k^*(w_k)$ for core k with partition factor w_k. In the **third step** (line 19 to 21), our algorithm iterates over all valid partitioning schemes to find the global optimal energy consumption. Line 20 gets the energy consumption for partition scheme P_j, and line 21 updates the final solution E^* with the minimal energy consumption. The time complexity for the first step is $O(m \cdot \rho_k \cdot \omega \cdot r^2)$, where m is the number of cores, ρ_k is the number of tasks on each core, ω is the number of ways in L2 cache, and r^2 is the number of L1 configurations. The time complexity for the second step is $O(m \cdot \omega \cdot \rho_k \cdot r^2 \cdot (e^{max} - e^{min}))$, where $e^{max} - e^{min}$ is the energy range. The time complexity for the third step is $O(m \cdot |\mathbf{P}|)$, where m is the number of cores and $|\mathbf{P}|$ is the number of partition schemes. In our experiments, our proposed approach can find the optimal solution within 8 hours (for 20 benchmarks and an error threshold of 5%). Almost all of it is spent on the profile table generation of the benchmarks. Since our approach is based on static (offline) analysis and one-time effort, this is a reasonable time.

In our proposed approach, the optimal partition factor is calculated and assigned statically to each core. Once the partition factor is assigned to a core, it is not changed on a per task basis. For a new set of tasks, the algorithm must re-run to get the new optimal L1 configurations and L2 partition factors. If a completely new task is introduced, then both the profiling and dynamic programming steps are necessary. If a task is not new but only changed in the task set to which it belongs, then we do not need to profile it again. Redoing only the dynamic programming step will suffice in this case.

5 EXPERIMENTS

In order to evaluate the effectiveness of our approach, we used the architectural simulator gem5 [4] in system emulation (SE) mode to simulate the multicore system as shown in Figure 3. We enhanced the simulator to support reconfiguration of L1 caches and way-based partitioning of the shared L2 cache. We also embedded our measurement for vulnerability of caches in the simulator, while the energy estimation of the cache subsystem is calculated with a script after simulation. Training of the machine learning model for the profile table generation is done using the *Statistics and Machine Learning Toolbox* of Matlab 2017b [1]. We configured our system with a four-core processor running at 500MHz on each core with the TimingSimpleCPU model in gem5. The shared L2 cache supports 32KB, 8-way associative with 32-byte lines. There are 35 valid schemes to partition the L2 ways among the four cores. The L1 caches have a base configuration as 4KB, 2-way associative with 32-byte lines, which offers an effective size of 1KB, 2KB, and 4KB, and associativity of 1-way, 2-way, and 4-way, and a line size of 16-byte, 32-byte, and 64-byte. There are 18 configurations in total for the L1 caches.[3] We used 20 applications from the MiBench [10] and SPEC CPU2000 [14] benchmark suites as our tasks for evaluation. Table 3 shows the task sets used in our experiments. We choose four task sets that contain two tasks running on each core, three task sets that contain three tasks on each core, and two task sets that contain four tasks on each core. The task assignment on cores is based on the rule that each core will have comparable execution time and vulnerability.

In our experiments, we have compared the following four approaches:

- **CP only:** The base configuration, which has L1 in base configurations and uniform L2 cache partitioning among cores.
- **DCP+CP[48]:** The energy-aware approach in Ref. [48] using DCR on L1 and CP on L2.
- **Vulnerability-aware[19]:** Vulnerability-aware energy optimization approach using DCR on L1 and CP on L2.

[3]It is fewer than 3^3 since not all combinations are valid [47].

Table 3. Task Sets from the MiBench [10] and SPEC CPU2000 [14] Benchmarks

Task set	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Set 6**	**Set 7**	**Set 8**	**Set 9**
Core 1	qsort vpr	mcf sha	applu lucas	mgrid FFT	mcf toast sha	mgrid parser gcc	vpr sha FFT	sha mcf untoast toast	gcc stringsearch parser dijkstra
Core 2	parser toast	gcc bitcount	dijkstra swim	dijkstra parser	gcc parser stringsearch	toast FFT mcf	CRC32 lucas untoast	applu gcc bitcount ammp	untoast mcf ammp bitcount
Core 3	untoast swim	patricia lucas	ammp FFT	CRC32 swim	patricia qsort vpr	bitcount ammp applu	mgrid bitcount qsort	lucas FFT CRC32 patricia	lucas patricia qsort vpr
Core 4	dijkstra sha	basicmath swim	basicmath stringsearch	applu bitcount	basicmath CRC32 ammp	qsort dijkstra patricia	applu parser stringsearch	vpr basicmath mgrid swim	basicmath toast applu CRC32

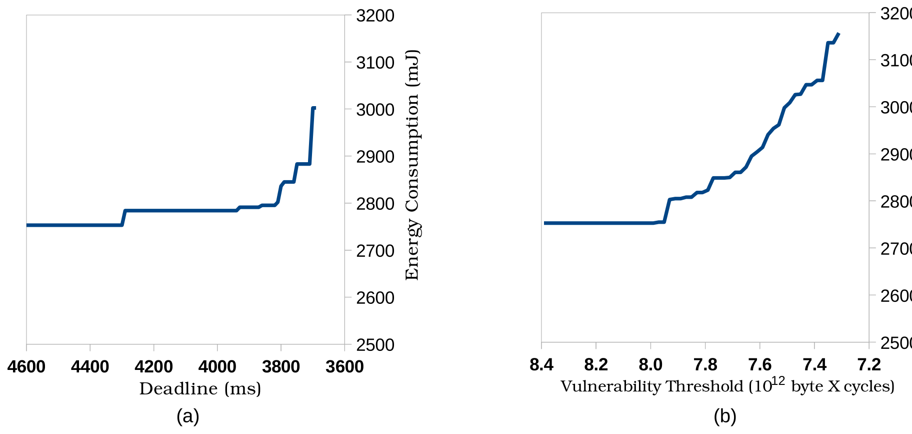

Fig. 9. Effects of deadline and vulnerability threshold on optimal energy consumption.

—**This work:** Replaces exhaustive simulation-based task profiling of Ref. [19] with machine learning model.

Here, **CP Only** refers to the base configuration of the system, which has uniform L2 cache partitioning among the four cores with all the L1 caches in base configuration. For our vulnerability-aware approach, the vulnerability threshold on each core is set as that of the base system (**CP Only**). We want to minimize the energy consumption while ensuring that the vulnerability is comparable or better than the base system.

5.1 Deadline and Vulnerability Threshold

It is meaningful to see how deadline and vulnerability threshold affect the optimization process. Figure 9 shows the optimal energy consumption (i.e., $E_1^*(w_1)$ as in Equation (10)) of core 1 using partition factor ($w_1 = 2$) for task set 9, under different deadline and vulnerability constraints. As expected, if we reduce the deadline, the tasks need to finish faster to meet the deadline. Thus, cache configurations with lower deadlines are then selected by our algorithm, even if these configurations have high energy consumption. Therefore, the total energy consumption goes up as the deadline decreases. Similarly, energy consumption increases for more stringent vulnerability thresholds.

Table 4. Effect of Error Thresholds on Required Training Data Size and Static Profiling Time

Error Threshold (%)	Average Training Data (%)	Test Error (%)	Actual Error (%)	Simulation Time[1] (s)	Training Time (s)	Prediction Time (s)	Total Time (s)	Speedup Over [19]
1	33.25	0.80	0.48	84,319	5282	<1	89,601	2.83
2	22	1.52	0.97	55,790	2384	<1	58,174	4.36
3	16	2.43	1.51	40,575	1328	<1	41,903	6.05
4	12.5	3.11	2.04	31,699	1043	<1	32,742	7.75
5	10.75	3.54	2.46	27,261	935	<1	28,196	8.99
6	9.5	4.07	3.00	24,091	853	<1	24,944	10.17
7	8.75	4.52	3.39	22,189	802	<1	22,991	11.03
8	8.25	4.92	3.54	20,921	776	<1	21,697	11.69
9	7.5	5.7	4.09	19,019	736	<1	19,755	12.84
10	7.5	5.7	4.09	19,019	736	<1	19,755	12.84

[1]Calculated from exhaustive simulation time and percent training data used.

Specifically, in Figure 9(a), as we gradually vary the deadline from 4,600ms to 3,600ms, the optimal energy found by the dynamic programming algorithm will become worse. When the deadline is shorter than 3,690ms, there is no feasible solution. In Figure 9(b), as we gradually reduce the vulnerability threshold from 8.4×10^{12} to 7.2×10^{12} bytes-cycles, the optimal energy solution will also become worse. There is no solution when the vulnerability threshold is set smaller than 7.3×10^{12} bytes-cycles. In this example, we can get a converged optimal energy solution (2,753mJ) with a deadline larger than 4,300ms and a vulnerability threshold larger than 8.0×10^{12} bytes-cycles. Note that in Figure 9(a), we removed the vulnerability constraint (i.e., set vulnerability threshold as infinity) to solely investigate the effect of deadline and vice versa for Figure 9(b).

This example suggests that the choice of deadline and vulnerability threshold can affect the optimal energy solution. In our experiments, the deadline is chosen in a way so that each core can reach the converged minimum energy under the base configuration setting. The vulnerability threshold on each core is also the same as the base system that runs with uniform L2 partition and the base configuration for L1s. These settings are performed under the assumption that our approach should not be more vulnerable than the base system while improving the energy profile. This assumes that our system should not be more vulnerable than the base system. In other words, we want our energy optimization process to be vulnerability-aware.

5.2 Profile Table Generation

As described in Section 4.1, we have used an iterative machine learning approach for profile table generation. Our experimental setup has 18 L1 data and instruction cache configurations and a maximum L2 partition factor of 8. With 20 benchmarks, the total number of profile table entries thus becomes $18 * 18 * (8 - 1) * 20 = 45{,}360$. We have collected 5% data in each iteration until the test error becomes smaller than the error threshold. Table 4 summarizes the results of task profiling. The first column shows the error threshold. The second column gives the average training data required to meet the error threshold. Here, training data is expressed as a percentage value. This is the ratio of profile table entries filled using simulation and total number of profile table entries, averaged over the 20 benchmarks. The third column gives the error on the test dataset, Y_{test}. This test error is used to check if the error threshold criteria is satisfied. Consequently, this value must be lower than the error threshold. The fourth column gives the actual error, which is calculated on Y_{all}. This value is essentially lower than the test error, because error for the simulated entries (Y_{sim}) would be zero. All errors are given in NRMSE measurement. The fifth column gives the

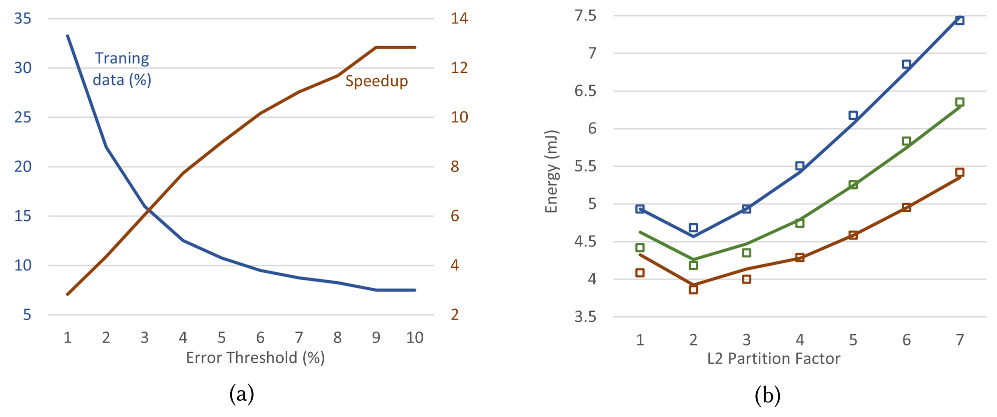

Fig. 10. (a) Required training data for different error thresholds and corresponding speedup. Here, training data is the average across all the benchmarks. (b) Actual vs. predicted energy values for *parser* benchmark. Three different configurations are shown; 10% of entries are used for training. Solid lines are actual values, and square boxes are the prediction.

estimated simulation time, t_{sim_train}. Here, the total simulation time (t_{sim_all}) for the complete profile table is known, and the estimate is derived from the percent of the training data used. The sixth column gives the training time to build the model (t_{train}). The seventh column shows the time required to predict values from the trained model (t_{pred}). As we can see, the experimental results confirm our previous assumption of $t_{sim\_train} \gg t_{train} \gg t_{pred}$. The eighth column shows the total time. The last column gives the speedup over the exhaustive simulation approach [19]. Static profiling time is not reported in Ref. [19]. We have simulated for all 45,360 configurations to get this time, which is 253,592sec. As we can see, our approach can provide more than 10× speedup with as little as 3% prediction error. If we allow only 1% error, then the speedup will be around 3×. This indicates that the proposed approach can provide substantial speedup even when high accuracy is crucial.

Figure 10(a) shows the total amount of training data required and the speedup for different error thresholds (columns 2 and 9 of Table 4). This graph can be used as a guideline to select the initial error threshold depending on the simulation time budget. Figure 10(b) demonstrates the actual and predicted energy values for *parser* benchmark with three randomly selected configurations. The model is trained with only 10% data. In this graph, solid lines represent actual values, and the same colored square boxes represent the predicted values. We can visualize that the predicted values closely follow the actual values obtained from simulation.

Figure 11 provides the training data requirement for the benchmarks (15 out of the 20 benchmarks are plotted to avoid clutter). We can see that some benchmarks require a lot of training data for accurate predictions (e.g., CRC32, FFT, sha), while some benchmarks need much less (e.g., applu, gcc, mcf etc.). This observation forms the basis of using error threshold instead of fixing training data percentage. The error threshold based approach allows more simulation time to benchmarks that require more training data to be accurate. For example, if we fixed the training data to 40%, five of the benchmarks would have NRMSE over 1%. Using error threshold instead gives below 1% error for all benchmarks using a little over 33% training data.

Shallow neural network is used for training the models. Shallow neural network is a variant of neural network with a small number of hidden layers, usually only one. We also used one hidden layer in our models. In each iteration, hyper-parameter sweep is done to tune the number of nodes in that layer. During the sweep, node number is increased from 10 to 100 in a step size of 10, and

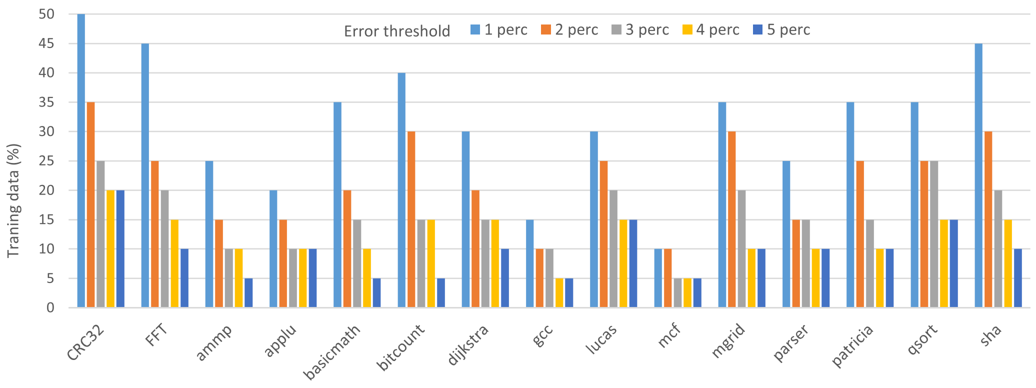

Fig. 11. Training data required with varying error threshold for different benchmarks. This graph is a more detailed view of Figure 10(a) for 1% to 5% error threshold.

Table 5. Hyper-parameter Tuning of Shallow Neural Network for Energy, Runtime, and Vulnerability Prediction

Benchmark	1% error threshold				5% error threshold				10% error threshold			
	Training data (%)	# of hidden nodes			Training data (%)	# of hidden nodes			Training data (%)	# of hidden nodes		
		Energy	Time	Vul		Energy	Time	Vul		Energy	Time	Vul
CRC32	50	40	40	50	20	20	30	20	15	20	20	20
FFT	45	50	50	40	10	10	10	30	5	20	20	20
ammp	25	30	50	30	5	10	40	20	5	10	40	20
applu	20	20	30	20	10	10	10	10	5	10	20	30
basicmath	35	30	20	30	5	20	30	20	5	20	30	20
bitcount	40	30	10	40	5	20	10	20	5	20	10	20
dijkstra	30	30	30	30	10	10	10	10	5	20	50	30
gcc	15	20	30	20	5	30	10	10	5	30	10	10
lucas	30	50	40	20	15	20	20	20	10	10	30	10
mcf	10	10	30	10	5	20	10	20	5	20	10	20

the model with least error is selected. As explained in Section 4.1, we have three models for each task—for predicting energy, runtime, and vulnerability. Table 5 shows the number of hidden layer nodes after hyper-parameter tuning. The first column is the name of the benchmark. Ten out of the twenty benchmarks are shown here. The second column shows the amount of data used for training the models. The third, fourth, and fifth columns give the number of hidden nodes for energy, runtime, and vulnerability prediction models, respectively. These values are shown for 1% error threshold. Columns 6–13 use similar notation for 5% and 10% error thresholds. As expected, the amount of required training data decreases for increasing error threshold. The number of hidden nodes for least error usually decreases with lower training data. This is because more nodes make the model prone to over-fitting. However, this tread does not always hold, making the parameter sweeping an effective way to find the most suitable number of nodes.

5.3 Vulnerability-aware Energy Reduction

Figure 12 illustrates the comparison of vulnerability and energy consumption of the nine task sets in Table 3. Here, the vulnerability is the maximum vulnerability among four cores while energy

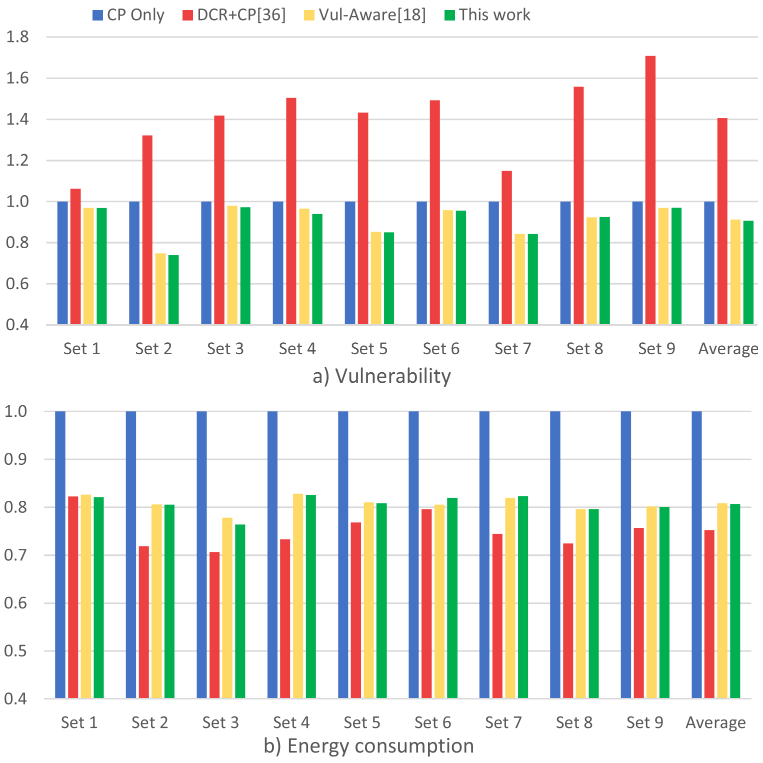

Fig. 12. Comparison of vulnerability and energy consumption for the cache hierarchy.

consumption is the total energy consumption of all L1 caches and L2 partitions. The maximum vulnerability provides an indication of the overall reliability of the cache subsystem since all the cores are independent with its private L1 caches and designated L2 partition.

Figure 12(a) shows the results for vulnerability reduction. Here, the yellow column is the normalized vulnerability with exhaustive task profiling [19]. The green column is our proposed approach with machine learning prediction based task profiling. We have used 5% error threshold for the machine learning models. As we can see, the prediction follows the actual optimal solution of Ref. [19] very closely, with maximum error of 1.78% for task set 6. On average, the error is less than 0.1% for vulnerability prediction. Even less deviation is achievable by setting tighter error thresholds, with the downside of more profiling time. Compared with **CP Only**, our approach reduces vulnerability by up to 25.2% and on average 8.8%. Compared with Ref. [48], our approach achieves up to 73.9% reduction in vulnerability and 49.3% on average.

Figure 12(b) shows the energy savings. Like vulnerability, our approach matches closely with energy consumption figures of the optimal exhaustive method of Ref. [19]. The maximum error is 2.69% for set 4, with average error of 0.56%. As we are using an error threshold of 5%, overall profiling time speedup over Ref. [19] is 9× (from Table 4). Compared with **CP Only**, our approach reduces energy consumption by up to 22.2% and 19.2% on average. Compared with Ref. [48], our approach consumes on average 5.6% and up to 9.5% more energy. In summary, our vulnerability-aware energy optimization can significantly reduce energy (on average 19.2%) compared with the base system. Compared with the state-of-the-art approach for energy optimization, we gain significant vulnerability reduction (on average 49.3%) with minor energy overhead (on average 5.6%).

Table 6. Task Set 1: Cache Config. ($[c_I, c_D, w_k]$)

Set 1	**Core 1** $w_1 = 2$	**Core 2** $w_2 = 2$	**Core 3** $w_3 = 1$	**Core 4** $w_4 = 3$
Task 1	[4KB_4W_16B, 2KB_2W_32B] **qsort**	[2KB_2W_64B, 4KB_4W_16B] **parser**	[2KB_2W_32B, 2KB_2W_16B] **untoast**	[2KB_2W_64B, 2KB_2W_16B] **dijkstra**
Task 2	[1KB_1W_64B, 4KB_4W_16B] **vpr**	[4KB_1W_64B, 1KB_1W_16B] **toast**	[4KB_4W_32B, 2KB_2W_32B] **swim**	[1KB_1W_64B, 1KB_1W_32B] **sha**

Table 7. Task Set 9: Cache Config ($[c_I, c_D, w_k]$)

Set 9	**Core 1** $w_1 = 2$	**Core 2** $w_2 = 2$	**Core 3** $w_3 = 2$	**Core 4** $w_4 = 2$
Task 1	[1KB_1W_64B, 2KB_2W_16B] **gcc**	[1KB_1W_64B, 1KB_1W_16B] **untoast**	[4KB_4W_16B, 2KB_2W_32B] **lucas**	[1KB_1W_64B, 4KB_4W_16B] **basicmath**
Task 2	[4KB_1W_32B, 4KB_4W_16B] **stringsearch**	[1KB_1W_32B, 1KB_1W_16B] **mcf**	[1KB_1W_64B, 1KB_1W_16B] **patricia**	[4KB_1W_64B, 1KB_1W_16B] **toast**
Task 3	[2KB_2W_64B, 4KB_4W_16B] **parser**	[1KB_1W_64B, 1KB_1W_16B] **ammp**	[4KB_4W_16B, 2KB_2W_32B] **qsort**	[1KB_1W_64B, 1KB_1W_16B] **applu**
Task 4	[2KB_2W_64B, 2KB_2W_16B] **dijkstra**	[1KB_1W_32B, 1KB_1W_32B] **bitcount**	[1KB_1W_64B, 4KB_4W_16B] **vpr**	[2KB_1W_32B, 2KB_2W_16B] **CRC32**

In order to understand the rationale for the above improvement, we would like to analyze the optimal solutions returned by Algorithm 1 for two different tasks sets. Tables 6 and 7 show the results of L2 partition factors and [IL1, DL1] cache configurations found by our approach for task set 1 and task set 9, respectively. Task set 1 has two tasks on each core, with a partition scheme of [2, 2, 1, 3] ways dedicated for each core. Task set 9 has four tasks on each core, with a partition scheme of [2, 2, 2, 2]. We can see that different tasks have very different L1 configurations, which shows the necessity of DCR to suit the unique needs of a task. For a certain task, the best [IL1, DL1] configurations depend not only on the task itself (i.e., its data access patterns), but also the L2 partition factor as well as the deadline and vulnerability threshold. There are a few tasks appearing in both Set 1 and Set 9. For benchmarks *qsort*, *vpr*, *parser*, and *toast*, they have the exact same L2 partition factor and L1 configurations for the two sets. For benchmark *untoast*, Set 1 and Set 9 have chosen different L1 configurations when Set 1 (Core 3) uses a partition factor of 1 and Set 9 (Core 2) uses a partition factor of 2. Because Set 9 assigns a larger partition factor, *untoast* can execute with smaller L1 cache sizes ([1KB, 1KB]) for reducing energy under the deadline and vulnerability constraints.

Vulnerability-constrained systems can tolerate up to certain vulnerability level due to its implemented mitigation solution. Therefore, existing energy-optimization techniques (such as Ref. [48]) are not applicable on them. For example, if a system can tolerate up to 20% more vulnerability compared to the base configuration, most of the energy savings (except for Set 1 and

Set 7) are meaningless since they crossed the vulnerability threshold. In other words, the apparent energy benefit of Ref. [48] is not useful in practice. Therefore, our vulnerability-aware energy optimization approach is vital for multicore systems with vulnerability constraints.

6 CONCLUSION

Cache vulnerability is a major concern in embedded systems design due to increasing cache size and soft errors. While both vulnerability and energy optimization have received considerable attention in recent years, there are no existing works on vulnerability-aware energy optimization for multicore systems. In this article, we presented a vulnerability-aware energy optimization technique for real-time multicore systems. Our approach integrates DCR of private L1 caches and CP of the shared L2 cache. L2 CP is effective in reducing inter-core interference, while applying L1 DCR can further reduce the energy consumption under the performance and vulnerability constraints. Our task profiling technique based on machine learning can reduce the exploration time by an order-of-magnitude. Our proposed algorithm uses dynamic programming by discretizing the energy values, which can efficiently search the space to find optimal L1 cache configurations for each task and L2 cache partition factors for each core. Experimental results demonstrated that we can achieve 19.2% average energy savings compared with the base system, while drastically reducing the vulnerability (49.3% on average) compared to the existing approaches.

REFERENCES

[1] Statistics and Machine Learning Toolbox, Matlab. 2017b. MathWorks, Retrieved from https://www.mathworks.com/products/statistics.html.

[2] Tosiron Adegbija, Ann Gordon-Ross, and Arslan Munir. 2012. Dynamic phase-based tuning for embedded systems using phase distance mapping. In *IEEE 30th International Conference on Computer Design (ICCD'12)*. IEEE, 284–290.

[3] G-H Asadi, V. S. Mehdi, B. Tahoori, and David Kaeli. 2005. Balancing performance and reliability in the memory hierarchy. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*. IEEE, 269–279.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[5] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee, and Ram Rangan. 2005. Computing architectural vulnerability factors for address-based structures. *ACM SIGARCH Comput. Archit. News* 33, 532–543.

[6] Yuan Cai, Marcus T. Schmitz, Alireza Ejlali, Bashir M. Al-Hashimi, and Sudhakar M. Reddy. 2006. Cache size selection for performance, energy and reliability of time-constrained systems. In *2006 Asia and South Pacific Design Automation Conference*. IEEE Press, 923–928.

[7] Shounak Chakraborty and Hemangee K. Kapoor. 2016. Static energy reduction by performance linked dynamic cache resizing. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'16)*. IEEE, 1–6.

[8] Timothy J. Dell. 1997. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division* 11 (1997).

[9] Ann Gordon-Ross and Frank Vahid. 2007. A self-tuning configurable cache. In *Proceedings of the 44th Annual Design Automation Conference*. ACM, 234–237.

[10] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization (WWC-4'01)*. IEEE, 3–14.

[11] Hadi Hajimiri, Prabhat Mishra, Swarup Bhunia, Branden Long, Yibo Li, and Rashmi Jha. 2013. Content-aware encoding for improving energy efficiency in multi-level cell resistive random access memory. In *IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'14)*. 76–81.

[12] Hadi Hajimiri, Kamran Rahmani, and Prabhat Mishra. 2011. Synergistic integration of dynamic cache reconfiguration and code compression in embedded systems. In*International Green Computing Conference (IGCC'11)*. 1–8.

[13] Hadi Hajimiri and Prabhat Mishra. 2012. Intra-task dynamic cache reconfiguration. In *25th International Conference on VLSI Design (VLSID'12)*. IEEE, 430–435.

[14] John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35.

[15] Po-Yang Hsu and TingTing Hwang. 2013. Thread-criticality aware dynamic cache reconfiguration in multi-core system. In *International Conference on Computer-Aided Design*. IEEE Press, 413–420.

[16] Hadi Hajimiri, Kamran Rahmani, and Prabhat Mishra. 2012. Compression-aware dynamic cache reconfiguration for embedded systems. *Elsevier Sustainable Comput. Inf. Syst.* 2, 2 (2012), 71–80.

[17] Yuanwen Huang and Prabhat Mishra. 2011. Vulnerability-aware energy optimization for reconfigurable caches in multitasking systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* https://ieeexplore.ieee.org/document/8355959.

[18] Yuanwen Huang and Prabhat Mishra. 2016. Reliability and energy-aware cache reconfiguration for embedded systems. In *17th International Symposium on Quality Electronic Design (ISQED'16)*. IEEE, 313–318.

[19] Yuanwen Huang and Prabhat Mishra. 2017. Vulnerability-aware energy optimization using reconfigurable caches in multicore systems. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 241–248.

[20] Reiley Jeyapaul and Aviral Shrivastava. 2011. Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors. In *14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 105–114.

[21] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy K. John. 2009. Bank-aware dynamic cache partitioning for multicore architectures. In *International Conference on Parallel Processing (ICPP'09)*. IEEE, 18–25.

[22] Yohan Ko, Reiley Jeyapaul, Youngbin Kim, Kyoungwoo Lee, and Aviral Shrivastava. 2015. Guidelines to design parity protected write-back L1 data cache. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC'15)*. IEEE, 1–6.

[23] Yun Liang, Xiuhong Li, and Xiaolong Xie. 2017. Exploring cache bypassing and partitioning for multi-tasking on GPUs. In *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 9–16.

[24] Sparsh Mittal, Yanan Cao, and Zhao Zhang. 2014. Master: A multicore cache energy-saving technique using dynamic cache reconfiguration. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* 22, 8 (2014), 1653–1665.

[25] Sparsh Mittal and Jeffrey S. Vetter. 2016. A survey of techniques for modeling and improving reliability of computing systems. *IEEE Trans. Parallel Distrib. Syst.* 27, 4 (2016), 1226–1238.

[26] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. 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'36)*. IEEE, 29–40.

[27] Xiaoke Qin and Prabhat Mishra. 2014. TECS: Temperature- and energy-constrained scheduling for multicore systems. *International Conference on VLSI Design (VLSID'14)*. 216–221.

[28] Xiaoke Qin, Weixun Wang, and Prabhat Mishra. 2012. TCEC: Temperature- and energy-constrained scheduling in real-time multitasking systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 31, 8 (2012), 1159–1168.

[29] Soft Error Rates. 2002. Neutrons from above. *Actel* (2002).

[30] Kamran Rahmani, Prabhat Mishra, and Swarup Bhunia. 2012. Memory-based computing for performance and energy improvement in multicore architectures. *ACM Great Lakes Symposium on VLSI (GLSVLSI'12)*. 287–290.

[31] Rakesh Reddy and Peter Petrov. 2010. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Trans. Embedded Comput. Syst. (TECS)* 9, 3 (2010), 16.

[32] Nathan N. Sadler and Daniel J. Sorin. 2007. Choosing an error protection scheme for a microprocessor's L1 data cache. In *International Conference on Computer Design (ICCD'06)*. IEEE, 499–505.

[33] Alex Settle, Dan Connors, Enric Gibert, and Antonio González. 2006. A dynamically reconfigurable cache for multithreaded processors. *J. Embedded Comput.* 2, 2 (2006), 221–233.

[34] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, Vol. 31. ACM, 336–349.

[35] Vilas Sridharan, Hossein Asadi, Mehdi B. Tahoori, and David Kaeli. 2006. Reducing data cache susceptibility to soft errors. *IEEE Trans. Dependable Secure Comput.* 3, 4 (2006), 353–364.

[36] Vilas Sridharan and Dean Liberty. 2012. A study of DRAM failures in the field. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE, 1–11.

[37] Jinho Suh, Mehrtash Manoochehri, Murali Annavaram, and Michel Dubois. 2011. Soft error benchmarking of L2 caches with PARMA. *ACM SIGMETRICS Perform. Eval. Rev.* 39, 1 (2011), 85–96.

[38] Weixun Wang, Sanjay Ranka, and Prabhat Mishra. 2011. A general algorithm for energy-aware dynamic reconfiguration in multitasking systems. *International Conference on VLSI Design (VLSID)*, 334–339.

[39] Weixun Wang, Prabhat Mishra, and Ann Gordon-Ross. 2009. SACR: Scheduling-aware cache reconfiguration for real-time embedded systems. *International Conference on VLSI Design (VLSID)*, 547–552.

[40] Weixun Wang and Prabhat Mishra. 2010. Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems. *International Conference on VLSI Design (VLSID)*, 357–362.

[41] Weixun Wang and Prabhat Mishra. 2009. Dynamic reconfiguration of two-level caches in soft real-time embedded systems. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'09)*. IEEE, 145–150.

[42] Weixun Wang, Xiaoke Qin, and Prabhat Mishra. 2010. Temperature- and energy-constrained scheduling in multitasking systems: A model checking approach. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 85–90.

[43] Weixun Wang, Sanjay Ranka and Prabhat Mishra. 2011. Energy-aware dynamic reconfiguration algorithms for real-time multitasking systems. *Elsevier Sustainable Comput. Inf. Syst.* 1, 1 (2011), 35–45.

[44] Weixun Wang and Prabhat Mishra. 2012. System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* 20, 5 (2012), 902–910.

[45] Weixun Wang and Prabhat Mishra. 2011. Dynamic reconfiguration of two-level cache hierarchy in real-time embedded systems. *J. Low Power Electron.* 7, 1 (2011), 17–28.

[46] Weixun Wang, Sanjay Ranka and Prabhat Mishra. 2011. Energy-aware dynamic slack allocation for real-time multitasking systems. *Elsevier Sustainable Computing: Informatics and Systems (SUSCOM)*, 2, 3 (2012), 128–137.

[47] Weixun Wang, Prabhat Mishra, and Ann Gordon-Ross. 2012. Dynamic cache reconfiguration for soft real-time systems. *ACM Trans. Embedded Comput. Syst.* 11, 2 (2012), 28.

[48] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. 2011. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *Proceedings of the 48th Design Automation Conference*. ACM, 948–953.

[49] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. 2012. *Dynamic Reconfiguration in Real-Time Systems*. Springer.

[50] Chuanjun Zhang, Frank Vahid, and Walid Najjar. 2005. A highly configurable cache for low energy embedded systems. *ACM Trans. Embedded Comput. Syst.* 4, 2 (2005), 363–387.

[51] Wangyuan Zhang, Xin Fu, Tao Li, and José Fortes. 2007. An analysis of microarchitecture vulnerability to soft errors on simultaneous multithreaded architectures. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'07)*. IEEE, 169–178.

Received May 2018; revised October 2018; accepted January 2019