# Smart Cache Cleaning: Energy Efficient Vulnerability Reduction in Embedded Processors

Reiley Jeyapaul and Aviral Shrivastava
Compiler Microarchitecture Lab
Arizona State University, Tempe, Arizona, USA.
{Reiley.Jeyapaul, Aviral.Shrivastava}@asu.edu

## ABSTRACT

Incessant and rapid technology scaling has brought us to a point where todays, and future transistors are susceptible to transient errors induced by energy carrying particles, called *soft errors*. Within a processor, the sheer size and nature of data in the caches render it most vulnerable to electrical interferences on static data in the cache. Data in the cache is *vulnerable* to corruption by soft errors, for the time it remains in the cache. Write-through and early-write-back [17] cache configurations reduce the time for vulnerable data in the cache, at the cost of increased memory writes and therefore energy. We propose a smart cache cleaning methodology, that enables copying of only specific vulnerable cache blocks into the memory at chosen times, thereby ensuring data cache protection with minimal memory writes. Our experiments over LINPACK and Livermore benchmarks demonstrate 26% reduced energy-vulnerability product compared to that of hardware cache configurations.

## Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS; C.3.3 [**Computer Systems Organization**]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS—*Real-time and embedded systems*

## General Terms

Reliability

## Keywords

soft error, vulnerability, hybrid technique, smart cache architecture, energy efficient, cache write-back

## 1. INTRODUCTION

Continuous technology scaling provides us with the capability to fabricate complex functionality, into smaller processor chips, consuming low-power, and at affordable costs. As a result, the use of embedded systems built with such processors have exploded, with them being used in many application areas not imagined before. This includes, medical, automotive, security systems, and in- and out-of-body sensing devices. On the other hand, a consequence of rapid technology scaling has been that, the transistors have become more fragile and susceptible to *soft errors*. Soft errors are transient faults that can occur due to one or more of several reasons, like electrical noise, external interferences, cross-talk, etc. However, majority of the soft errors in digital devices happen due to charge-carrying particle strikes on the processor that corrupt its logic value. Such corruption of data used within the processor may lead to system failure. Charge carrying particles, like alpha-particles and high energy neutrons (of 100KeV 1GeV from the cosmic background), have been known to cause soft errors in semiconductor devices for a long time [18]. With technology scaling, even low energy neutron particles (of 10meV 1eV) can cause soft errors in sub 45nm SRAM memory cells [25]; which is exacerbated by the fact that there are many more low-energy neutrons, than those with higher energies [10]. At the current technology node, high-end embedded systems, e.g., smart-phones, tablets, etc., incur a Soft Error Rate (SER) of about *once-per-year*, but is expected to increase exponentially with technology scaling [13]. With embedded systems finding use in several safety-critical applications, the importance of protecting them from soft errors cannot be overstated. Protecting embedded systems from soft errors is not easy, as any protection scheme will have some power and/or performance overheads; they are crucial concerns for embedded systems. As a result, power-efficient soft error protection techniques are required for embedded systems.

In a processor, the cache is most susceptible to soft errors. This is not only because it occupies majority of the chip area, but also because it has a high transistor density; it is again, operated at lower supply voltages, reducing the critical charge ($Q_{crit}$) required to flip a stored data-bit [21]. Estimated soft error rates of typical designs such as microprocessors, network processors, and network storage controllers, show that unprotected SRAMs contribute to more than 40% of the overall soft error rate. As a consequence of technology scaling, the size of the on-chip cache increases steadily with each generation [26]. Since reliability of memory elements (SRAM cells) is projected to remain constant for the recent future (the positive impact of smaller bit areas will be offset by the negative impact of storing lesser charge per bit), the cache error rate as a whole will increase linearly with cache size [8]. To model the susceptibility of data in caches, the metric of *vulnerability* is used [24]. A datum is vulnerable (or susceptible to data corruption by soft errors) in the cache, only if it is dirty (written by the processor), *and* is then either, i) read by the processor, or ii) written back to the next level of memory. Herein, the assumptions of the underlying cache architecture are that:

**i)** the probability of double-bit errors is negligible (typically 3 orders of magnitude lesser [20]), in comparison to single-bit errors; simple hardware techniques like interleaving the bits of a cache-line can reduce the onset of multi-bit errors.

**ii)** data in the cache is protected by parity bit error detection [7] (as in popular processor architectures like Intel Xscale® [11], Intel IA-32® [12], AMD Athlon® [1], etc.).

In a cache where every cache-line is protected by parity bits, if an error is detected on a cache-line which is not dirty (.i.e., clean or not updated), it can be invalidated and reloaded from the memory as a cache-miss. One method to protect cache-data from soft errors, is by ensuring that an updated copy of all the cache-data is available in the memory (to re-load, when an error is detected). A write-through cache ensures such a scenario, by writing copies of cache-blocks as and when they are updated in the cache, thereby realizing *zero* vulnerability. However, write-through caches suffer from very high memory traffic between the cache and the rest of the memory subsystem. These memory-writes keep the data-bus busy, thereby increasing the cache-miss latency for new memory accesses, and affect the overall performance of the system. Another consequence is excess energy consumed by the memory subsystem:

**i)** at the data-bus between the cache and the lower levels of memory (which are typically off-chip in embedded systems), and

**ii)** by accesses to the lower level memory components on every write-back; increasing the total power consumption of the system.

To find a middle ground, Early Write-Back (EWB) [17] cache architecture was proposed; In this, all the dirty cache-blocks are written back to the next level of memory only at periodic intervals. Reducing the frequency of write-backs, reduces the memory traffic and therefore the power consumption of the system, but at the cost of cache-data vulnerability, when compared to a write-through cache. By varying the periodicity of cache write-backs, EWB caches can explore the inversely proportional trade-off between vulnerability reduction and power overhead due to the additional memory traffic.

Both these techniques write-through (WT), and early write-back (EWB) are hardware techniques, and are not sensitive to the data access patterns of the application. For example, if a datum is vulnerable across two write-back periods (in the EWB technique, by executing write-backs at designated intervals), the additional write-backs eventually do not affect the vulnerability realized, but only increases memory traffic. If the cache write-back process can be customized according to the changing data access pattern of applications, vulnerability reduction can still be achieved, but with reduced number of write-backs, and thus reduced power overheads. Thus, there is scope for power-efficient vulnerability reduction by customizing the write-backs based on the data access patterns of data in a program.

In this paper, we propose a hardware-software hybrid scheme: Smart Cache Cleaning (SCC), that provides a means to dynamically moderate the cache write-backs and thus achieve power-efficient vulnerability reduction in embedded systems. Our scheme is composed of three important components:

1. application analysis to determine, which data accessed in the program, has to be write-back and when the said write-back has to be executed, to achieve power-efficient vulnerability reduction,

2. succinctly represent the time sequence identified as to when a reference must be written back, and

3. transfer this information to the specialized SCC architecture, that performs write-backs of the specified references at specified times.

Our experiments over scientific benchmark loops like Livermore [2] and LINPACK [19] show that smart cache cleaning achieves 26% better energy-vulnerability product than the lowest energy-vulnerability product achieved by the EWB scheme (across various write-back periods). Our SCC scheme achieves almost *zero* vulnerability, at $< 1\%$ power overhead. Using the EWB scheme, to achieve the same level of vulnerability, a minimum of 40% and an average of 2.88× power-overhead is incurred.

## 2. DATA CACHE VULNERABILITY

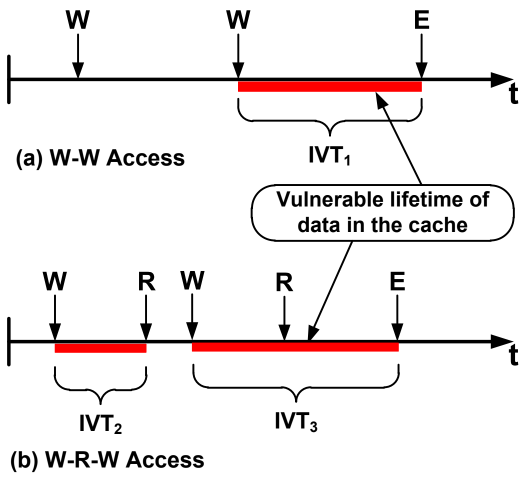


**Figure 1: Intermediate Vulnerable Time (IVT) .i.e., the time a data element remains vulnerable in the cache is defined for two data access patterns (where, W=Write, R=Read, E=Eviction): a data element once written is vulnerable as long as it remains in the cache across read accesses. Since the second write (W) operation over-writes the updated data element, any error that may affect the unused data in the cache, deems the access as *not-vulnerable* for that time slot.**

On a store operation from the processor, a data element in the cache (byte or word) is written. The containing cache-line now is deemed dirty, such that this becomes the only updated copy of the data stored. The time that such dirty cache-data remain in the cache, it is vulnerable to data corruption by soft errors. Over the sequence of different accesses on the cache-data (write (W), read (R), eviction or write-back (E)), the vulnerability of the data varies based on its usage. We define *Intermediate Vulnerable Time (IVT)*, as the duration for which a data element is vulnerable in the cache; after being updated by a write operation. Data access patterns in a program can be broadly classified into two types WW (write only) and WR (write and read). In Fig. 1, the two data access patterns over a data element are portrayed, and the IVT definition in each case is highlighted:

a) The updated (written) data may be over-written by the program. In this case, since the updated data (in the first write operation) is not used but again updated, any soft error on the stored data between the two write accesses is not recognized. Therefore, the IVT for the data here is only the time from the last write operation to the time it was evicted (E) from the cache; when the data updates the underlying memory.

b) The updated (written) data may be used by other data accesses (read) during the course of the program. Since the correctness of this data is essential for the correct functioning of the system, it is vulnerable throughout this duration in the cache (till eviction E). However, as in Fig. 1(b), if the same data is updated by another write operation, the data is over-written; therefore, the time slot between the last use (read) and update (write) opertion is deemed *not-vulnerable*.

## 3. MOTIVATION

We motivate the need for a mechanism to dynamically moderate the cache write-backs, with help of an example as in Fig. 2(a). For this, we take a two-dimensional loop operating over two arrays, executing for a total of nine iterations. We assume that, during the course of the program, there occur no cache-evictions or write-backs due to cache-line replacements; the program terminates on the $10^{th}$ iteration or cycle and all the cache-data is finally written-back to the memory. We again assume that the cache is protected with a parity-bit error detection mechanism, such that its copy from the memory can be re-loaded, if an error is detected on the cache-data. From the definition of *vulnerability*, we observe that the three elements of array `B` are only read, and therefore not vulnerable. On the other hand, the three elements of array `A` are updated (read and written) on every iteration, and therefore vulnerable for the entire time that they remain in the cache. In Fig. 2(c), a time-line for the program execution is drawn and below it, in each section is the cache behavior on the elements of array `A`, in each cache write-back configuration. Immediately below the time-line in Fig. 2(c), the position of each element of array `A` denotes the time it is first accessed by the program (for e.g., `A[2]` is first accessed when index `i`=2 and `j`=1 on the $4^{th}$ iteration of the program); the vertical dotted line that follows, indicates the last iteration it is updated/used by the program.

In the baseline write-back cache configuration, once an element is loaded into the cache, it is not evicted by any additional write-backs. Therefore the data element remains vulnerable from its first access (write access) till the end of the program; described by red bars, for each element, extending from the start to end of its life-time in the cache. The total number of write-backs (wb) and the vulnerability (vul) of each data element is marked on the right of the time-line. In the write-through cache configuration, on each iteration when the data in the cache is updated, it is also written-back to the memory. The downward pointing arrows in Fig. 2(c) denote the write-back operations in the cache, on every iteration of the program; which is a characteristic of the write-through cache configuration. Rightly so, the vulnerability of data in the cache is 0, because there always exists a copy of the updated data in the lower level memory (which can be re-loaded when an error is detected in the cache), and data in the cache is always *clean*.

On similar lines, we explore the vulnerability vs write-back count ratio of the early write-back (EWB) cache architecture and our customized smart cache cleaning scheme. Li et.al. [17], report through design space exploration the power efficiency trade-offs involved in the choice of a write-back period. Based on their recommendations and using a conservative estimate for the sample program and cache model considered, we set the write-back period to be 4 iterations. In this, once every 4 iterations of the program, the EWB architecture identifies *dirty* data in the cache and writes-back the same into the lower-level memory, thus rendering the data *clean*. We observe that this mechanism achieves 55% reduction in data-cache vulnerability, at the cost of only 22% additional write-backs (compared to the WT cache)to the lower-level memory. The highly regular nature of the sample program here, ensures such a profit by this technique, but such is not the case in general purpose applications. We arrive at such a conclusion owing to some key observations on the operation of the EWB scheme:

1. The *pre-defined periodic nature of the write-backs* rarely corroborate with the data access patterns of the application. Cache-data here, more often than not, tend to remain vulnerable beyond the time they are required and/or updated by the program. For example, in Fig. 2(c) the double-ended arrows indicate the time that each data element remained vulnerable, after it was last updated by the program. This functionality of the EWB scheme, causes the array `A` to be, vulnerable for an additional 4 unused iterations.

2. The working of the EWB scheme is to *identify all dirty cache-lines on each period and perform write-backs on all of them*, may require writing-back (or cleaning) data when it can be used/updated by the program in the immediate future. For example, in Fig. 2(c), during the access time of `A[3]`, a periodic write-back (once every 4 iterations) cleans `A[3]` in addition to the previously accessed `A[2]`. However, since `A[3]` is updated the very next iteration (which in-turn is the last iteration it is used), the data remains vulnerable from then till the end of the program. This functionality of the EWB scheme, while reducing vulnerability by *one* iteration, causes the data to remain vulnerable for *one* additional iteration; the additional vulnerable time would at the least be *four*, if the program runs for more than 10 iterations.

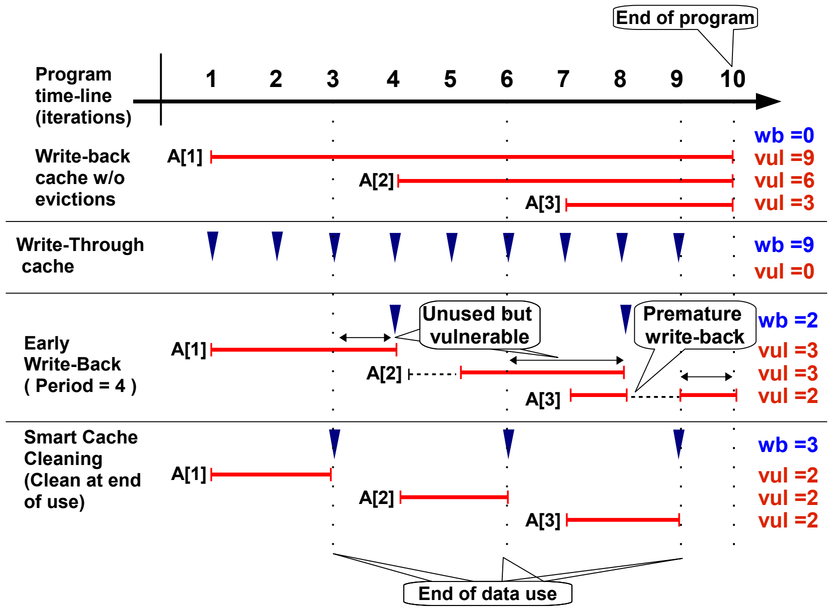
In Fig. 2(c), the last section below the time-line, describes the vulnerability of each data element and the number of write-backs required in our smart cache cleaning technique. Here, we observe 67% vulnerability reduction, with only

```
for(i : 1 to 3){
  for(j : 1 to 3){
    A[i]+= B[j];
  }
}
```

(a) Example program

| Cleaning Period | Vulnerability of Array A | Additional write-backs |
|---|---|---|
| No WB | 18 | 0 |
| WT | 0 | 9 |
| EWB 4 | 8 | 2 |
| **SCC** | **6** | **3** |

(b) Tabulated program statistics



(c) Cache write-backs and its vulnerability impact

**Figure 2: Demonstrating the need and importance of smart cache cleaning. (a) Two dimensional loop operating over two data arrays one read-only (`B`) and the other read-and-written (`A`). (b) Summary of array `A`'s vulnerability in each cache configuration shows the SCC scheme achieves energy efficient vulnerability reduction. (c) Detailed iteration-level analysis of cache write-backs, in each configuration, and their impact on the vulnerability of array `A`'s elements.**

one additional write-back (compared to the EWB scheme). Here, i) data is vulnerable in the cache only for as long as it is used; and ii) every write-back operation is timed and positioned in such a way that it achieves overall vulnerability reduction. Such an adaptive scheme, that can dynamically moderate the cache-write-backs, would thus achieve power efficient vulnerability reduction on cache-data.

## 4. RELATED WORK

Careful selection and screening of materials [3], SOI fabrication technologies [5], increasing the transistor size or adding gated resistors [22] are some hardware techniques proposed to reduce soft errors in SRAM cells. In addition to the chip area and power overheads, the cost of design and fabrication of such device/circuit level techniques, and the yield obtained upon manufacture, over-weighs the reliability achieved. This reduces its applicability and/or wide-spread use of such methods to protect embedded systems. At the architecture level, ECC based techniques like SECDED [9] provide a means to protect the caches by storing ECC codes for every cache block and checking the same for correctness when used by the processor. In this, 8 *check bits* are required for every 64 bits of cache-data, which involves a 12.5% increase in the size of the cache. In addition to this, additional logic, to generate and verify the ECC codes of the read/written data, is added to the cache read-write path. Li et.al [16] in their work indicate that the hardware costs (area, performance and power) incurred in the implementation of such an ECC based error detection and correction technique is unacceptable for embedded systems. Sridharan et al [26], propose selective re-fetching of cache lines combined with a write-through cache implementation to achieve 85% reduced cache vulnerability at the cost of 2.5% power, 7% performance and 15% chip area overheads. Authors in [17], propose to use a fixed interval early write-back technique to periodically clean the dirty cache lines and reduce their vulnerable lifetime. In spite of the reduced hardware overhead involved in its implementation, such a technique has been shown to achieve an effective trade-off between vulnerability and power only for large caches (hundreds of MB or GBs). Zhang [27] in his work, proposes two hardware based techniques (LRU and Dead-time based prediction scheme), to vary the periodic interval between write-backs from the L1 cache to the underlying memory. In this, the methodology used does not acknowledge the availability of 1-bit parity based error detection hardware in almost all modern processors, thereby underusing the available resources. In addition, for the implementation for such a smart hardware only scheme, the additional hardware required would add to the additional write-backs executed, thereby adding to the total hardware performance and energy costs to the system. In this work, we aim to achieve increased reliability in a system, by utilizing the available resources in the system, with minimum additional hardware, performance and/or energy costs. We also show through experiments over varying range of periods, that such a hardware technique when implemented in an embedded processor, has a significant impact on the number of cache write-backs and thereby adds to the power consumption of the system and also affects performance.

Software solutions are preferred as they can be implemented on existing architectures. The authors in [24] develop Cache Vulnerability Equations (CVE) to determine statically the vulnerability of a program for a particular cache configuration. We motivate on this understanding of data reuse and cache vulnerability to develop a profile analysis techniques to determine important store references and accesses that have to be cleaned to achieve vulnerability reduction with reduced memory writes.

Software-hardware hybrid techniques have the advantage of reduced architecture overhead and the flexibility and accessibility to hardware structures aided by software techniques. Chen et al [6], propose a compiler based technique to determine the critical data used in the application and enable error correction techniques(ECC) for only those data elements. Partially Protected Caches (PPC) in which a portion of the cache is protected against soft errors can achieve around $47X$ vulnerability reduction in data intensive multimedia applications [15]. Lee et.al [14] then propose compiler techniques to statically partition data into critical and non-critical, to further enhance the protection available in a PPC. In this work, we determine the *right data* to clean and *exactly when* to do so through memory profile analysis, and with the help of hardware support, ensure that vulnerability reduction is achieved with reduced energy overhead.

## 5. OUR APPROACH

### 5.1 Key Idea

Copying a dirty cache block into the memory, through write-backs (cache cleaning), reduces the vulnerability of the system but incurs an energy overhead due to memory accesses. To reduce energy overheads while also increasing reliability in a system, a prudent decision has to govern each cache cleaning operation ensuring that a memory access is performed *iff* significant vulnerability can be reduced. In embedded applications, such prudence can be achieved through profile based techniques which help in identifying the right references and the right instances that cache-data has to be cleaned.

### 5.2 Overview

The memory profile information of a given program is used to evaluate the vulnerability per cache write-back profit metric for each data-reference (store instruction) in the program in the *Reference Selection* stage. For each reference in the list (*scc_reference* list) and a given threshold *scc_threshold* for the intermediate vulnerable time (IVT) generated during accesses, the list of instruction accesses to be cleaned are identified. This decision formed as a bit stream for each reference is represented by a $k$-bit pattern, where $k =$*scc_pattern_size*. The instruction addresses and their corresponding representative $k$-bit patterns (*scc_pattern*) are instrumented into the given program through compiler directives to be loaded into their respective hardware components accordingly. With the help of hardware support from the cache cleaning architecture, the embedded processor now executes the program with minimal additional cache write-backs, and maximum reliability.

### 5.3 Smart Cache Cleaning Architecture

The shaded blocks in Fig. 4 represent the hardware components added to implement our smart cache cleaning technique. The "SCC Register Pair" contain the instruction address (*scc_insn_addr*)and bit pattern (*scc_clean_pattern*), for the reference set to be cleaned by profile analysis. On every access to the targeted store instruction (marked as `csw` in the

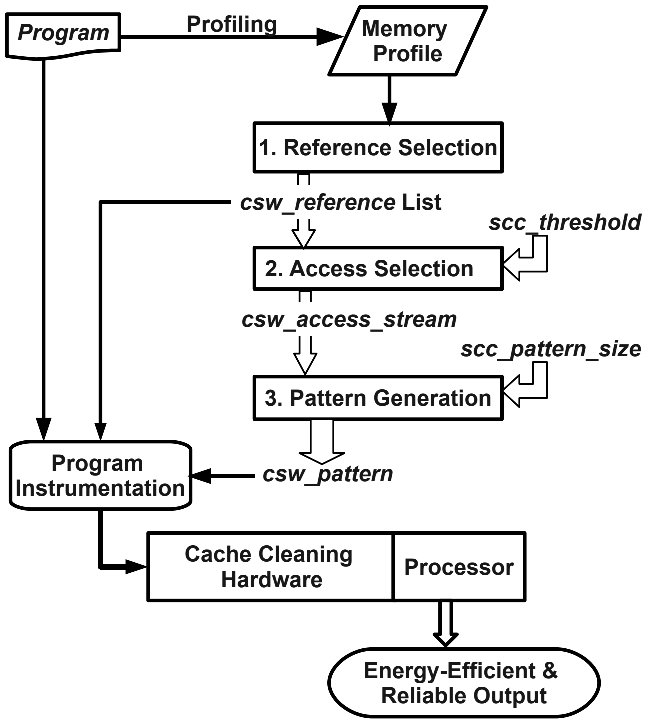


**Figure 3: Our 4 stage Smart Cache cleaning methodology.**

instrumented code), an bit-iterator iterates over the pattern in the *scc_clean_pattern* register. A 1 read by the iterator indicates (through the *Clean EN* signal) that the cache block accessed has to be cleaned (copied to the memory by a cache write-back) while a 0 indicates otherwise. The iterations on this *scc_clean_pattern* are wrapped such that the pattern is repeatedly accessed throughout the program runtime that the corresponding instruction is accessed.

The instrumented program input to the processor, contains special instructions to load SCC-data into the "SCC Register Pair" at specific points in the program based on the memory profile analysis. The remainder of this section describes in detail the 4 step procedure that generates SCC-data for program instrumentation (as shown in Fig. 3) and thereby trigger the cache cleaning architecture blocks to ensure energy efficient reliability. The "Targeted Cache Cleaning Block" performs the *cache cleaning* operation as follows,:

1. the target cache-block address to be cleaned is input along with the *Clean EN* signal, from the LSQ.

2. data from the specific cache block is copied, and written-back into the underlying memory. This operation is performed after the completion of the `sw` operation, and independent of the memory access thereby causing no interference to the cache performance of the system.

Overall the SCC architecture requires $1 \times (32 + 32) = 64$ bit register, $1\times$ 32bit XOR gate and $1\times$ 2bit AND gate. We assume here that these SCC registers are protected against soft errors by energy efficient hardware techniques for the same. We observe that the targeted cache cleaning architecture exists in most modern embedded processors in the form

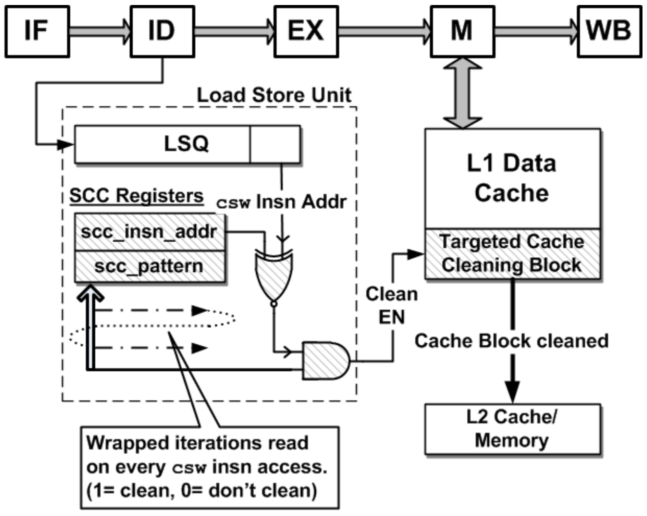


**Figure 4: Smart Cache Cleaning Architecture: The architecture blocks as part of the SCC are shaded. Every marked store instruction (denoted by `csw`) is compared and based on the cleaning decision read by the iterator, *Clean EN* is signaled triggering targeted cache block cleaning.**

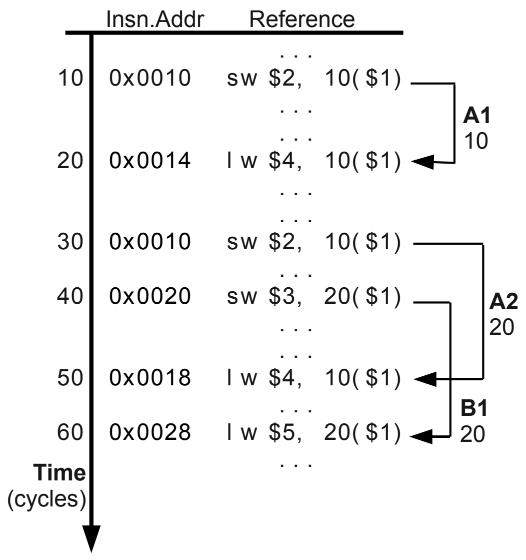


**Figure 5: Demonstrating Smart Reference Selection: On the memory profile of a program over its execution time-line, arrays $A$ and $B$ are accessed by instruction addresses $0x0010$ and $0x0020$ respectively. The individual IVT values (A1, A2, B1) are labeled and annotated by arrows that connect their W and R accesses points.**

of cache-flush execution units, which can be modified (if required) with minimal hardware changes. It is thus evident, that the overall area and power overheads of the additional hardware components required for our SCC implementation, are minimal and negligible.

## 5.4 Step 1: Smart Reference Selection

The memory profile data gathered by profiling the application is used to identify the set of references that generate significant vulnerability, and therefore have to be cleaned accordingly. The key idea behind this reference selection step is that it is possible to identify the vulnerability generated by each reference individually and thereby compare references based on the vulnerability per access metric. This metric gives an estimate of the possible vulnerability-energy trade-off that can be achieved if all the reference accesses are set to be cleaned.

Every store instruction during program execution may accesses different data elements, and each such accesses renders a cache block vulnerable. This time for which the accessed cache block remains vulnerable in the cache, is defined as the Intermediate Vulnerable Time (IVT) (defined in Section 2) generated by that instruction access. In order to map the vulnerability of a program to the references generating them, we calculate *ref-vulnerability* for each reference, defined as the sum of all the IVT values generated by the reference during program execution. The profit metric for each reference is thus given by $\frac{ref-vulnerability}{ref-access-count}$. From the description of our cache cleaning architecture in Fig. 4 we observe that the there is only one SCC Register Pair and therefore only one reference can be set to be cleaned at any point in time. Among the references in the program, the chosen set of references to be cleaned are those with highest vulnerability per access values (or highest profit), with non overlapping execution time-lines.

The program in Fig. 5 is of the W-R-W access pattern and the IVT values generated by reference $A$ is $A1$, $A2$ and that for reference $B$ is $B1$. From the annotated data in Fig. 5 we derive the data table Table 1. It can be noted here that, the efficiency achieved due to higher profit numbers, in selecting reference $B$ to be cleaned, automatically precludes selection of reference $A$ under the non overlapping execution time-lines condition.

| Parameters | Ref A | Ref B |
|---|---|---|
| ref-vulnerability | $10 + 20 = 30$ | 20 |
| ref-access-count | 2 | 1 |
| CSW_Profit | $\frac{30}{2} = 15$ | $\frac{20}{1} = 20$ |

**Table 1: Data table derived for statistics on the example program in Fig. 5.**

## 5.5 Step 2: Smart Access Selection

In a program, the *ref-vulnerability* generated depends not only on its own data access pattern but also is affected by other references and data elements accessed on a set associative cache, causing cache-block replacements (cache eviction). We thus understand that not all IVT values of a reference are same owing to possible cache replacements. Having identified the most profitable reference(s) to clean, the most profitable reference accesses can be identified as those that have IVT values greater than a given threshold value (design parameter *scc_threshold*). This threshold defines the maximum time (in cycles) that vulnerable data can remain in the cache before being evicted or overwritten. For every access by the selected reference, the IVT values generated (by each access) are compared with the given *scc_threshold*, and those that exceed the threshold are slated to be cleaned. This cleaning decision is represented by a bit stream (*scc_access_stream*)of length equal to the total number of accesses by the reference, and a *Clean* operation is denoted by 1 on the bit stream and a 0 otherwise. This stream in conjunction with the *scc_reference* list, thus contains input instructions for the smart cache cleaning architecture.

## 5.6 Step 3: Smart Pattern Generation

**Algorithm 1** SCC_BIT_PATTERN_MATCHING ()

```
Require: scc_access_stream <csw_list>, scc_clean_pattern_size K.
 1: for $k$ from 0 to K do
 2:     $k\_ones \leftarrow$ Count $1^s$ in csw_list
 3:     $k\_zeros \leftarrow$ Count $0^s$ in csw_list
 4:     $Cost\_of\_0 \leftarrow k\_ones \times 2$
 5:     $Cost\_of\_1 \leftarrow k\_zeros \times 1$
 6:     if $Cost_of_1 \leq Cost_of_0$ then
 7:         $BitPattern[k] = 1$
 8:     else
 9:         $BitPattern[k] = 0$
10:     end if
11: end for
12: return $BitPattern$
```

The bit stream (*scc_access_stream*) represents the set of accesses that have IVTs greater than a threshold for a reference that has been identified to have the highest vulnerability per access metric. In order to implement cache cleaning based on this bit pattern, multiple and complicated load instructions are required to ensure that the correct pattern is loaded into the *scc_clean_pattern* register for the corresponding instruction access. Therefore, a bit pattern of size $k$ (a design pattern defined by *scc_pattern_size*), has to be determined that best represents the bit stream *scc_access_stream* of the reference accesses. In line with our intentions to ensure smart energy efficient cache cleaning, we use the *SCC_Bit_Pattern_Matching* algorithm to analyze the bit stream and derive a representative $k$ bit pattern.

The *SCC_Bit_Pattern_Matching* algorithm described in Algorithm 1 reads the given bit stream and using a moving window of size $k$ bits, the number of $1's$ and $0's$ in each bit position are calculated. Using these numbers, a cost is associated ($Cost\_of\_0$ or $Cost\_of\_1$) with each bit position that represents the cost of representing the bit as 1 or 0 respectively. For example, for a given bit stream, if a particular bit position in the $k$ sized window, has many $0's$, it is right to assume that majority of these reference accesses don't generate vulnerability greater than the threshold and will therefore deliver low vulnerability savings for the energy cost, represented by the $Cost\_of\_1$ calculated. Therefore, giving precedence to energy savings, we ensure that a bit is represented by 1 *iff* the $Cost\_of\_1$ is less than twice the $Cost\_of\_0$ value. The costs associated with the bit positions thus ensures that the resultant $k$ bit pattern is an energy efficient representative of the given bit stream.

## 5.7 Step 4: Program Instrumentation and Execution

From the memory profile of a program, after the first 3 steps, a *scc_reference* list is identified, and then for each reference in this list, a representative $k$ bit pattern *scc_pattern* is generated. The program is then instrumented with these two inputs so as to instruct the processor hardware to load corresponding values into the "SCC Register Pair". Using the memory profile, access points of the first and last accesses for each reference in the list can be identified, and at these points, corresponding load instructions are introduced with the respective reference address and $k$ bit pattern data. It should be noted here that these instructions are compiler-directives and will not be executed through the processor pipeline, thereby involving negligible performance variation.

## 5.8 Cache Cleaning on Multiple References

Our smart cache cleaning architecture is scalable over the number of references set to be cleaned simultaneously. In the above discussion, we illustrate the use of only one SCC register pair (*scc_insn_addr*, *scc_clean_pattern*) while additional register pairs will enable the hardware to support multiple references to be cleaned over overlapping execution timelines. For this purpose, the only modification in the profile analysis will be, at step 1 where references are selected such that $n$ references may overlap in their execution time-lines, thereby allowing for corresponding access stream and $k$ bit pattern generation. It should be noted here that additional hardware structures involve additional area and power overheads ($32 + 32 = 64$ bits for each SCC register pair added), which does not add significantly to the existing architecture.

## 6. EXPERIMENTS AND RESULTS

## 6.1 Experimental Setup

For our experiments, we model an embedded system with a RISC processor, an on-chip L1 cache and off-chip SDRAM memory. The SimpleScalar [4] `sim-outorder` cycle-accurate simulator is configured to model the Intel XScale [11] processor architecture, with a 2-way set associative L1 cache (size = 4KB). The simulator is instrumented with code to accurately evaluate vulnerability of data used in the program (in byte cycles). To estimate memory access power, we use power numbers from the MICRON MT48V8M32LF SDRAM on an Intel 440MX chipset [23] to represent the off-chip components of the system. The energy per memory access is composed of data bus energy (9.46 nJ per burst) and SDRAM energy (32.5 nJ per read/write burst). The power consumed during memory accesses is given by the product of total number of memory accesses and the total energy per memory access (41.96 nJ). To experimentally demonstrate the effectiveness of our SCC methodology, the SimpleScalar `sim-outorder` simulator is modified to include the Smart Cache Cleaning Architecture blocks (described in Section 5) and also recognize our instrumented program instructions (`csw` instructions).

To compare the trade-off between vulnerability and energy on a one dimensional scale, we use the product of vulnerability (in byte-cycles) and memory access energy (in nJ) to form Energy Vulnerability Product (EVP). Here EVP provides us with a single metric to quantitatively compare the impact of the various configurations on both vulnerability and energy consumption, thereby allowing us to achieve the required balanced trade-off. In any application, the data acceses on arrays within nested loops, are the program segments that contribute to data-cache behavior. We perform our experiments on benchmark loops from *LAPACK* [2] and *LiverMore Loops* [19], which are scientific, data-intensive and computation-intensive benchmarks representative of applications executed on such embedded systems. To compile our benchmarks we used GCC (v 2.7.3) with all optimizations turned on. In our attempt to analyze the efficiency and impact of SCC, we experiment over each benchmark varying all the possible design parameters like *scc_threshold* $(5, 10, 15, \cdots, 200$ cycles), *scc_pattern_size* $(4, 8, 16, 32$ bits) and the number of *scc_insn_reg* registers (number of references to clean). We then compare the EVP values thus obtained with that of a write-through cache and early-writeback (EWB) cache configuration of varying periods $(100, 200, \cdots,$ 2000 cycles).

## 6.2 Better Energy-Vulnerability Efficiency With Smart Cache Cleaning

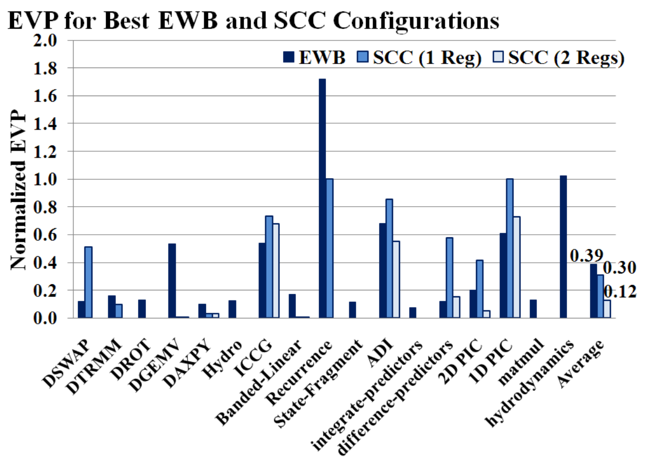


**Figure 6: The graph showing Normalized EVP of the best EWB period (EWB configuration with least EVP) and the best *scc_threshold* parameter using 1 and 2 *scc_insn_reg* registers, demonstrates higher energy-vulnerability efficiency with the SCC technique.**

The graph in Fig. 6 plots EVP values, normalized to that of the original program, obtained for the best EWB configuration and best SCC threshold values. For each benchmark, among the results obtained for the set of EWB periods experimented $(100, 200, \cdots, 2000$ cycles), we choose the one with the least EVP value. Similarly, from the results for varying *scc_threshold* values, we choose the threshold that delivers lowest EVP. The graph clearly demonstrates that overall the benchmarks, the SCC technique achieves lower EVP and therefore better energy efficient vulnerability reduction. From the graph in Fig. 6 we observe:

**(1)** The second bar (labeled ”SCC (1 Reg)”) represents normalized EVP values of the SCC technique obtained for experiments using 1 *scc_insn_reg* register, showing the efficiency obtained when only one reference is selected (to be cleaned) at any point in time. For most benchmarks this bar remains unseen owing to their significantly low EVP values $(\leq 2 \times 10^{-7})$ indicating highest possible efficiency.

**(2)** In the case of benchmarks like *ICCG, ADI, 2D PIC, 1D PIC* and *diff-predictor* the program contains multiple references executing in overlapping time-lines with comparable vulnerability per access profits, and therefore the selection of only one reference seems insufficient. For majority of the benchmarks experimented we observe that the use of a second *scc_insn_reg* register decreases the EVP significantly, which is represented by the third bar (labeled ”SCC (2 Reg)”).

**(3)** From the results for the *Recurrence* benchmark, we see that the early writeback mode of cache cleaning loses on EVP by 60% compared to the original program. Owing to the complex data access pattern in the program, the best early write-back configuration (EWB period = 1800 cycles) achieves only 26% vulnerability reduction at the cost of $2X$ increased memory writes. On the other hand, having the knowledge of the data access pattern and the flexibility to enable cache cleaning only at instances that achieve profitable vulnerability reduction, the SCC technique using one register achieves 26% vulnerability reduction at $< 1\%$ increase in memory writes. Moreover, with the use of an additional register, we achieve 100% vulnerability reduction at 96% increase in memory writes.

**(4)** The average EVP plots, towards the right end of the graph indicate that our SCC technique using one *scc_insn_reg* register is 8% lesser, and using two registers is 26% lesser than that of the EWB technique.

## 6.3 More Energy Efficient Design Points in SCC

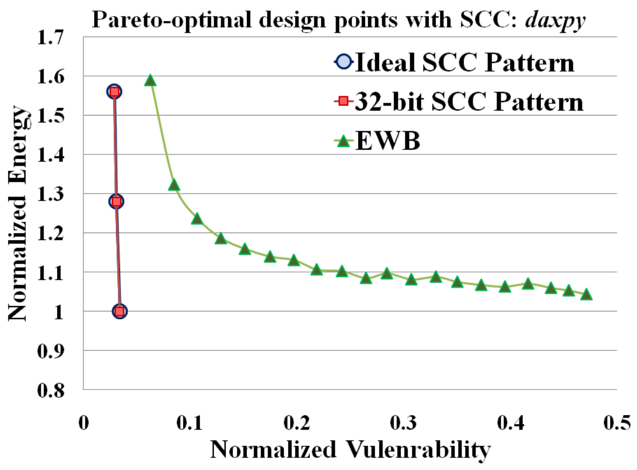


**Figure 7: Normalized vulnerability and energy plots for two benchmarks across varying *SCC_Threshold* values. The plots for *32-bit Pattern SCC* closely follow that by *Ideal SCC* in *DSWAP*, while they overlap in DAXPY, demonstrating the accuracy of Algorithm 1. Vulnerability and energy trade-off is observed for varying threshold values for each benchmark.**

For each benchmark, we experiment over varying thresholds and for each perform experiments using the ideal *SCC_Access_Stream* derived after memory analysis and again using the K-Bit pattern that best matches with the access stream (determined using the pattern matching algorithm Algorithm 1). In Fig. 7, The x-axis plots the normalized vulnerability values, while the y-axis plots normalized energy (number of cache write-backs). For one benchmark, in the EWB configuration, the early write-back period is varied $(100, 200, \cdots, 2000)$ and the normalized vulnerability and energy overhead incurred are plotted in Fig. 7 labeled EWB. Similarly, for the same benchmark the vulnerability and overhead values are plotted for varying threshold values $(5, 10\ , 15, \cdots, 200)$, using our SCC technique. In the graph plotted in Fig. 7, each plotted point is a design point in a design space exploration to determine the right EWB period or *SCC_Threshold* to choose for energy efficient vulnerability reduction. A design point closer to the x-axis denotes that it has a low energy overhead, and a point closer to the y-axis denotes low vulnerability (or increased reliability) of the program. A point that is close to the origin (0,0) is the most efficient point which denotes least vulnerability at least energy overhead. It can be clearly seen here, that the results from our SCC technique over varying threshold parameters have points more closer to the x-axis and also more closer to the y-axis than any other point in the EWB plot; thereby demonstrating the energy efficiency realized. In other words, we say that design points obtained by our SCC technique are pareto-optimal to design points achieved by hardware techniques like WT or EWB.

## 6.4 Generated K-bit pattern achieves close-to-ideal SCC efficiency

The algorithm *Generate_Bit_Pattern* defined in Algorithm 1, uses a weighted matching technique to analyze the ideal access stream (*SCC_Access_Stream*) of a reference selected to be cleaned, and represents the same as a k-bit pattern. In our experiments we evaluate the accuracy of the algorithm over *SCC_Pattern_Sizes* $4, 8, 16$ and 32. In Fig. 7, the normalized vulnerability and energy values for DAXPY, across varying threshold values, are plotted for a pattern size of 32bits. It is evident from the overlapping plots of SCC values in Fig. 7, the values obtained after pattern matching on a 32-bit register closely follow that of the ideal access stream (*CSW Access Stream*). This demonstrates the accuracy of our Smart Pattern Matching algorithm (Algorithm 1). We again observe that for larger pattern sizes, the extent of matching accuracy increases, but when implemented does not show any significant difference in the vulnerability and energy numbers. The system designer is thus able to choose between allowing one 32bit register or 2 16bit registers based on hardware constraints, and still achieve intended vulnerability reduction.

## 6.5 EVP decreases with increase in references to clean

When larger number of SCC registers are integrated into the system, our SCC technique provides for scalability and therefore energy efficient vulnerability reduction in the system. Fig. 8 plots the EVP values of four benchmarks for varying numbers of references selected to be cleaned simultaneously. For each benchmark, the maximum number of references allowed is determined through memory analysis. We observe here that, in each benchmark the small additional SCC registers, translate into significant EVP reduction. It is interesting to note that in the *Diff-Predictors* benchmark, with the choice of 2, 3 and 4 registers the greedy nature of selecting references to clean translates into greater EVP numbers, however as the number of selected references increases to 7, the EVP is significantly reduced.

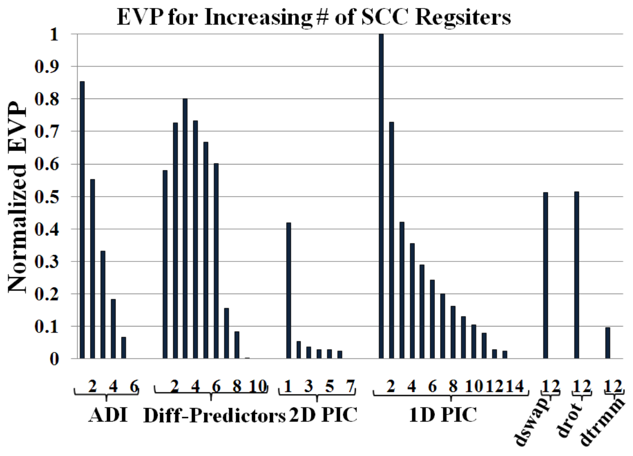


**Figure 8: The EVP of the application reduces with increase in the number of *scc_insn_reg* registers used.**

## 7. CONCLUSION

By reducing the time that vulnerable data resides in the cache, we can reduce the probability of an error in cache data and thereby reduce the overall system failure rate. Hardware based mechanisms like a write-through cache or an early write-back cache though efficient in reducing the vulnerable data time in the cache, incurs a large energy overhead due to increased L1-memory writes. we develop a hybrid hardware-software Smart Cache Cleaning (SCC) technique, where we use the memory profile of an application to accurately estimate data vulnerability (time that updated data is in the cache), identify the program instances that generate the same. We then enable cache cleaning on specific cache blocks at specific instances, to ensure energy efficient reduction of data cache vulnerability. Our experiments over scientific benchmarks show that when compared to the hardware based early write-back cache architecture, the SCC technique achieves 26% lower Energy Vulnerability Product.

## 8. FUTURE WORK

Our profile base method currently identifies references with higher profit on non-overlapping execution time-lines, but it is observed that for some applications, the references are accessed in bursts. In such a case, the use of a reference to clean can be interleaved with another to achieve better results. It is possible to analyze loops with affine access functions statically at the compiler for its vulnerability and thereby identify the right references and access instances to perform cache cleaning. Intelligent schemes can be devised to analyze the data access patterns statically, and thereby derive the design points for varying *threshold* and *k-bit* values. Such a methodology will help develop a well-rounded and automated scheme to implement smart cache cleaning.

## 9. ACKNOWLEDGMENTS

This work was partially supported by funding from National Science Foundation grants CCF-1055094 (CAREER), CCF-0916652, NSF I/UCRC for Embedded Systems (IIP-0856090), Raytheon, Intel, SFAz and Stardust Foundation.

## 10. REFERENCES

[1] AMD Corporation. AMD Athlon®Processor Product Data Sheet, 2007.

[2] E. ANDERSON. Lapack: Users' guide. 1995.

[3] R. Baumann, T. Hossain, S. Murata, and H. Kitagawa. Boron compounds as a dominant source of alpha particles in semiconductor devices. In *Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International*, pages 297 –302, apr. 1995.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[5] E. Cannon, D. Reinhardt, M. Gordon, and P. Makowenskyj. SRAM SER in 90, 130 and 180 nm bulk and SOI technologies. *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 300–304, April 2004.

[6] G. Chen, M. Kandemir, M. J. Irwin, and G. Memik. Compiler-directed selective data protection against soft errors. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 713–716, New York, NY, USA, 2005. ACM Press.

[7] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[8] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of cmos process scaling and soi on the soft error rates of logic processes. In *VLSI Technology, 2001. Digest of Technical Papers. 2001 Symposium on*, pages 73 –74, 2001.

[9] L. Hung, H. Irie, M. Goshima, and S. Sakai. Utilization of secded for soft error and variation-induced defect tolerance in caches. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1 –6, apr. 2007.

[10] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule. *Electron Devices, IEEE Transactions on*, 57(7):1527 –1538, July 2010.

[11] Intel Corporation. Intel XScale®Technology Overview, 2000.

[12] Intel Corporation. Intel IA-32®Developer's Manuals, 2007.

[13] S. Kayali. Reliability considerations for advanced microelectronics. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, PRDC '00, pages 99–, Washington, DC, USA, 2000. IEEE Computer Society.

[14] K. Lee, A. Shrivastava, N. Dutt, and N. Venkatasubramanian. Partitioning techniques for partially protected caches in resource-constrained embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 15:30:1–30:30, October 2010.

[15] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 17:1343–1347, September 2009.

[16] J.-F. Li and Y.-J. Huang. An error detection and correction scheme for rams with partial-write function. In *Memory Technology, Design, and Testing, 2005. MTDT 2005. 2005 IEEE International Workshop on*, pages 115 –120, Aug 2005.

[17] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Soft error and energy consumption interactions: a data cache perspective. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 132 – 137, Aug 2004.

[18] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2 – 9, Jan. 1979.

[19] F. McMahon. L. L. N. L. Fortran Kernels Test: MFLOPS, 1993.

[20] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *PRDC '04: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, pages 37–42, Washington, DC, USA, 2004. IEEE Computer Society.

[21] R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski. Critical charge characterization for soft error rate modeling in 90nm sram. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1879 –1882, May 2007.

[22] L. R. Rockett Jr. Simulated SEU hardened scaled CMOS SRAM cell design using gated resistors. *Nuclear Science, IEEE Transactions on*, 39(5):1532–1541, Oct 1992.

[23] A. Shrivastava, I. Issenin, and N. Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 90–96, New York, NY, USA, 2005. ACM.

[24] A. Shrivastava, J. Lee, and R. Jeyapaul. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 143–152, New York, NY, USA, 2010. ACM.

[25] C. Slayman. Alpha particle or neutron ser-what will dominate in future ic technology, 2010.

[26] V. Sridharan, H. Asadi, M. B. Tahoori, and D. Kaeli. Reducing data cache susceptibility to soft errors. *IEEE Transactions on Dependable and Secure Computing*, 3(4):353–364, 2006.

[27] W. Zhang. Computing and minimizing cache vulnerability to transient errors. *IEEE Des. Test*, 26:44–51, March 2009.