Enabling Energy Efficient Reliability in Embedded Systems Through Smart Cache Cleaning

REILEY JEYAPAUL and AVIRAL SHRIVASTAVA, Compiler Microarchitecture Lab,
Arizona State University

Incessant and rapid technology scaling has brought us to a point where today's, 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 interference on data stored in the cache. Data in the cache is *vulnerable* to corruption by soft errors, for the time it remains actively unused in the cache. Write-through and early-write-back [Li et al. 2004] cache configurations reduce the time for vulnerable data in the cache, at the cost of increased memory writes and thereby 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. In this work, we first propose a hybrid (software-hardware) methodology. We then propose an improved software solution that utilizes cache write-back functionality available in commodity processors; thereby reducing the hardware overhead required to implement smart cache cleaning for such systems. The parameters involved in the implementation of our Smart Cache Cleaning (SCC) technique enable a means to provide for customizable energy-efficient soft error reduction in the L1 data cache. Given the system requirements of reliability, power-budget and runtime priority of the application, appropriate parameters of the SCC can be customized to trade-off power consumption and L1 data cache reliability. Our experiments over LINPACK and Livermore benchmarks demonstrate 26% reduced energy-vulnerability product (energy-efficient vulnerability reduction) compared to that of hardware based cache reliability techniques. Our software-only solution achieves same levels of reliability with an additional 28% performance improvement.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*; C.1.4 [**Processor Architectures**]: Parallel Architectures—*Mobile processors*; C.3 [**Special-Purpose and Application-Based Systems**]: Microprocessor/Microcomputer Applications; C.4 [**Performance of Systems**]: Design Studies

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Reliability, soft error, power efficiency, embedded system, smart cache, cache cleaning

ACM Reference Format:
Jeyapaul, R. and Shrivastava, A. 2013. Enabling energy efficient reliability in embedded systems through smart cache cleaning. ACM Trans. Des. Autom. Electron. Syst. 18, 4, Article 53 (October 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2505012

This article is an extension of the conference paper "Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors" In Proceedings of the CASES '11.
This work was partially supported by funding from National Science Foundation grants CCF-0916652, CCF-1055094 (CAREER), NSF I/UCRC for Embedded Systems (IIP-0856090), Raytheon, Intel, SFAz and Stardust Foundation.
Author's address: R. Jeyapaul; email: reiley.jeyapaul@asu.edu.

DOI: http://dx.doi.org/10.1145/2505012

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 exponentially increased in quantity, with them being used in many application areas not imagined before including 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 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 interference, 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 [May and Woods 1979]. With technology scaling, even low energy neutron particles (of 10meV 1eV) have been identified to cause soft errors in sub 45nm SRAM memory cells [Slayman 2010]; which is exacerbated by the fact that there are many more low-energy neutrons, than those with higher energies [Ibe et al. 2010]. At the current technology node, high-end embedded systems, for instance, smart phones, tablets, etc., incur a Soft Error Rate (SER) of about once-per-year, but is expected to increase exponentially with technology scaling [Kayali 2000]. 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 [Naseer et al. 2007]. 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 [Sridharan et al. 2006]. 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 [Hareland et al. 2001]. To model the susceptibility of data in caches, the metric *vulnerability* is used [Shrivastava et al. 2010]. 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, and our soft-error model, are as follows.

(i) The region of coverage for our soft-error scheme is the L1 data cache, and system failure is modeled by the following case. (a) A soft-error on data consumed by the program, will result in program failure (not-withstanding cases when specific computations may mask the errors, for instance, multiplied by 0). (b) A soft-error on a dirty cache-line which is written-back into the memory, corrupts the underlying memory subsystem (which forms part of the program output), and therefore contributes to system failure.
(ii) The probability of multibit errors is negligible (typically 3 orders of magnitude lesser [Mukherjee et al. 2004]), in comparison to single-bit errors; simple hardware

techniques like interleaving the bits of a cache-line can reduce the onset of multibit errors.

(iii) Data in the cache is protected by parity bit error detection [Hamming 1950] (as in popular processor architectures like Intel Xscale® [2000], Intel IA-32® [2007b], AMD Athlon® [2007a], etc.).

In a cache where every cache-line is protected by parity bits, if an error is detected on a cache-line that is not dirty (i.e., clean or not updated), it can be invalidated and reloaded from the memory as a cache-miss. In this light, we recognize that the instruction cache is not vulnerable, based on our soft error model. One method to protect the cache-data from soft errors, is by ensuring that an updated copy of all the data in the cache is available in the memory (to reload, 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 demonstrating *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)[Li et al. 2004] cache architecture was proposed. In this, all the dirty cache-blocks are written back to the next level of memory only at periodic intervals. Reduced write-back frequency reduces the memory traffic and therefore the additional power consumption in the system, but at the cost of cache-data reliability; which is significantly higher 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 (based on the data-cache locality over space and time) 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 scheme where periodic write-backs are triggered), the additional write-backs eventually do not affect the vulnerability of the data, but only increase memory traffic. If the cache write-back process can be customized according to the changing data access pattern of the application, 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 article, 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 written 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.

The hardware components required for SCC implementation (as in Figure 5) in the processor pipeline is relatively small and of low energy overheads. However we recognize that the key hardware components required for SCC implementation include a targeted cache write-back block, which is readily available in modern commodity embedded processors (e.g., ARM, StrongARM, XScale, x86). In this journal, we develop a software-only solution to implement smart cache cleaning that utilizes the available processor architecture effectively. In this, an additional instruction (`csw` clean store word) is added into the ISA of the processor, which is a wrapper instruction to trigger the use of the architecture specific cache write-back instruction available in the processor's ISA. In this, the hardware functionality available in the hardware will have to be configured to recognize the newly added instruction and perform targeted cache-cleaning appropriately; which requires negligible hardware modifications. Through the use of loop unrolling code transformations, the chosen store instructions are converted into `csw` instead of the regular `sw` instruction. In this, in addition to storing the register data into memory, the stored cache block is also written-back into the memory; thus performing cache cleaning for the accessed cache-block.

Our experiments over scientific benchmark loops like Livermore [Anderson 1999] and LINPACK [McMahon 1993] show that smart cache cleaning achieves 26% better energy-vulnerability product (energy efficient vulnerability reduction) than the lowest energy-vulnerability product achieved by the EWB scheme (across various write-back periods). We perform design space experiments to demonstrate the influence of the various design choices (scc-threshold, cache size, k-bit size) involved in the generation of a SCC pattern to perform smart cache cleaning in an application. Our SCC scheme achieves almost zero vulnerability, for 1.2% area and 0.56% power overheads. On the other hand, using the EWB scheme to achieve the same level of vulnerability a minimum of 40% area and an average of 2.88× power-overhead is incurred [Li et al. 2004]. Our software-only compiler solution achieves an additional 28% performance improvement (in conjunction with code transformations), for the same reliability as that of our hybrid technique.

The energy consumption of the system can be monitored at runtime through the use of: (1) temperature sensors (the superlinear dependence of power over temperature makes this a viable means to monitor power without the need for additional specialized hardware sensors in the processor) (2) power state of the system as monitored and controlled by the operating system. Provided the existence of such mechanisms, the parameters involved in the design of our SCC pattern for a given application, can be customized and multiple SCC-patterns (or SCC triggers) can be generated using the memory profile information. At runtime, the appropriate SCC-pattern can be loaded into the system based on the power requirements, as it is possible to trade-off power for data cache reliability, and thus prevent further exacerbation of the power-hungry memory traffic. Thus, in this work, we discuss the use of SCC as a methodology to provide data cache reliability through design parameters that enable customizable energy-vulnerability trade-offs.

2. DATA CACHE VULNERABILITY

Mukherjee et al. [2003] define vulnerability and develop a systematic approach to estimate system AVF (Architecture Vulnerability Factor). Our definition of the *vulnerable* time periods of data in the cache is but an extended implementation of that defined in Mukherjee et al. [2003] (as ACE and un-ACE bits). Here, we define a means to identify vulnerable time-periods of cache-blocks during program execution; thereby developing a mechanism to perform smart cache cleaning on targeted data cache blocks.

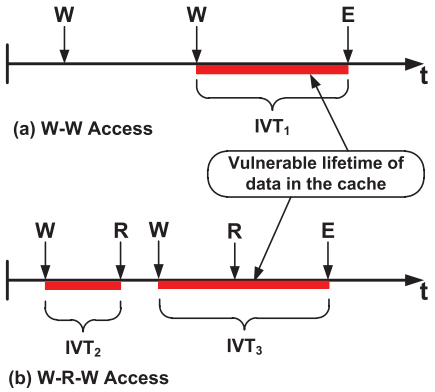


Fig. 1. Intermediate Vulnerable Time (IVT) that is, the time a data element remains vulnerable in the cache is defined for two data access patterns (where, W=Write, R=Read, E=Eviction): a data element once written is vulnerable as long as it remains in the cache across read accesses. Since the second 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.

2.1. Definition

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. In this, we don't differentiate between speculative and committed read operations on the data in the cache; therefore, the vulnerability measure estimated is a tight upper bound on the actual system vulnerability. Based on our analysis, our assumption does not affect the accuracy of the total data cache vulnerability estimated. 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. As a result of the write-back (E) cache event data is written into the underlying memory which, when corrupted, may eventually result in erroneous program execution/output; and therefore a dirty cache-block is vulnerable till the write-back cache event for that block. Data access patterns in a program can be broadly classified into two types WW (write only) and WR (write and read). In Figure 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 overwritten 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 Figure 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) operation is deemed *not-vulnerable*.

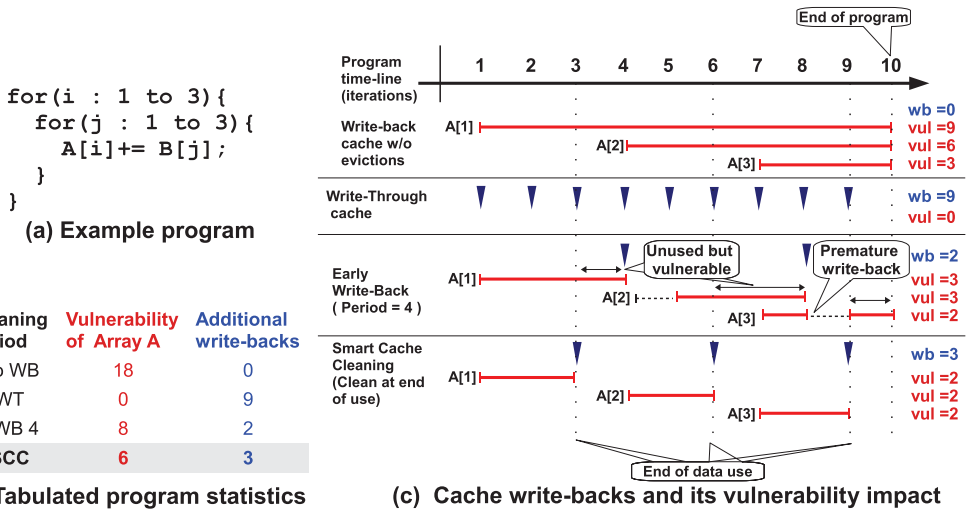


Fig. 2. Demonstrating the need and importance of smart cache cleaning. (a) Two-dimensional loop operating over two data arrays one read-only (B) and the other read-and-written (A). (b) Summary of array A's vulnerability in each cache configuration shows the SCC scheme achieves energy efficient vulnerability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

3. MOTIVATION

We motivate the need for a mechanism to dynamically moderate the cache write-backs, with help of an example as in Figure 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 reloaded, 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 Figure 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 Figure 2(c), the position of each element of array A denotes the time it is first accessed by the program (for instance, 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 lifetime 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 Figure 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 reloaded 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. [2004], 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 predefined 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 Figure 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 Figure 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 inturn 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 Figure 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 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.

3.1. Quantifying the Claims

In order to quantify the claims stated before, we performed an experiment on the `matmul` program, and observed the variation in vulnerability and the number of memory writes with each of the loop orders of the program over the same set of data. From the results gathered as in Figure 3, we observe that the vulnerability of the data in the program cannot be studied independent of the data access pattern of the program, and therefore any method to mitigate this "data-cache vulnerability," cannot be implemented

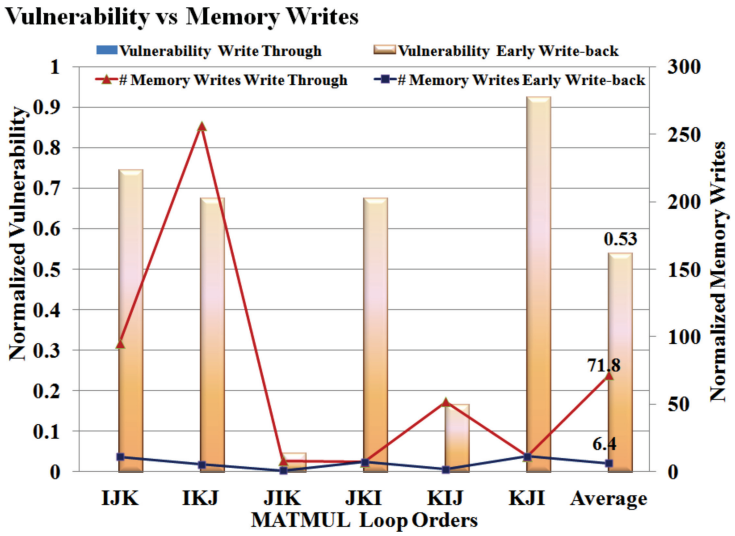


Fig. 3. Relative comparison of vulnerability (in byte-cycles) and the number of memory-writes incurred during the implementation of write-through cache, and early-writeback on a simple matrix multiplication program. The vulnerability in the presence of early-writeback varies with the loop order, which demonstrates the room for smart cache cleaning on programs by analyzing the data access patterns.

efficiently without appropriate analysis of the program's data access patterns. We propose Smart Cache Cleaning (SCC), in this work, to perform this specific data cache analysis and present an energy efficient mechanism to reduce data cache vulnerability in embedded systems.

4. RELATED WORK

Careful selection and screening of materials [Baumann et al. 1995], SOI fabrication technologies [Cannon et al. 2004], increasing the transistor size or adding gated resistors [Rockett Jr. 1992] 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 [Hung et al. 2007] 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 and Huang [2005] 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. [2006] propose selective refetching 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. Li et al. [2004] propose using 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 [2009] 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 under-using the available resources. In addition, for the implementation for such a smart hardware only scheme, the additional hardware overhead incurred 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 (early write-back scheme) 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. The primary goal of this work is to achieve power-efficient vulnerability reduction, and therefore as a comparison we use the early write-back scheme, which has a lower power consumption than that of the other hardware only techniques.

Software solutions are preferred as they can be implemented on existing architectures. Shrivastava et al. [2010] 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. In this journal, we present a software-only approach to implement smart cache cleaning, that utilizes the existing hardware resources and ISA extensions available in most modern processors. Through this, we achieve vulnerability values similar to that of the hybrid technique, but with negligible area and power overheads.

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. [2005] 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 [Lee et al. 2009]. Lee et al. [2010] 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 SMART CACHE CLEANING 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

To begin, the memory profile of the given program, is extracted using a cycle-accurate simulation of the application executed on the target architecture. This gives the exact

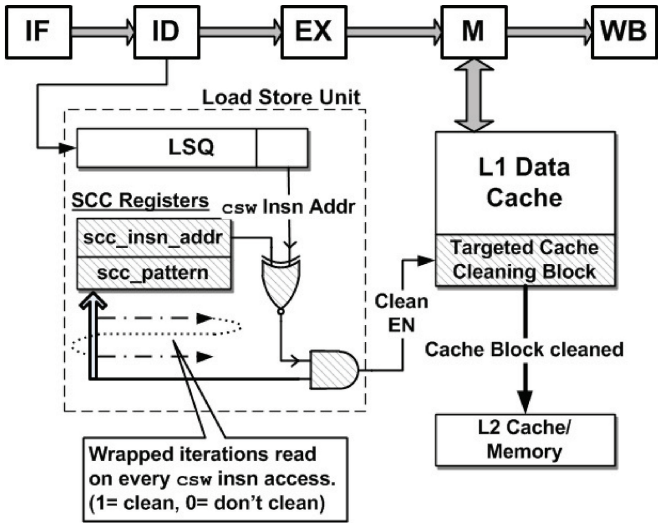


Fig. 4. Smart Cache Cleaning Architecture: The architecture blocks as part of the SCC are shaded. Every marked store instruction (denoted by csw) is compared and based on the cleaning decision read by the iterator, *Clean EN* is signaled triggering targeted cache block cleaning.

sequence and number of memory accesses in the program, and therefore provides means to perform exact analysis on the data cache vulnerability induced by the program's data access pattern. The memory profile of the program is fed as input to the *SCC Analysis* block, which gives us the required input to be fed into the *SCC Hardware* blocks for reliable and power efficient execution. 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.

The *SCC Analysis*, which answers the three key questions required to achieve the said energy-efficient reliability, is described as a three-stage procedure.

(1) *Which data to clean*? Using the memory profile information, the vulnerability induced per cache-access (*Instantaneous Vulnerable Time* (IVT)) for the data used in the program is computed. In order to avail energy efficient reliability, the key is to trigger cache write-backs after specific write operations on the cache-data, such that the larger IVTs induced by the write accesses can be eliminated. A profit metric is then generated to compare the energy efficient vulnerability reduction (high vulnerability low write-backs) possible by each reference, and thus the top references to clean are identified (*scc_reference* list.
(2) *When do we clean the selected data*? Once the reference is selected, the instances at which the SCC cleaning needs to be triggered, is a result of threshold based evaluation over the IVTs of each cache write access for that reference. The specific instances to clean are now identified (*scc_access_stream*.
(3) *How do we clean the selected data at the appointed times*? The resulting bit stream from the previous step is unmanageable by hardware and therefore has to be represented as a manageable bit pattern (*scc_pattern*) that represents accurately, the right instances where the reference needs to be cleaned; when replicated throughout the lifetime of the program.

5.3. Smart Cache Cleaning Hardware Architecture

The shaded blocks in Figure 4 represent the hardware components added to implement our smart cache cleaning technique. The "SCC Register Pair" contain the

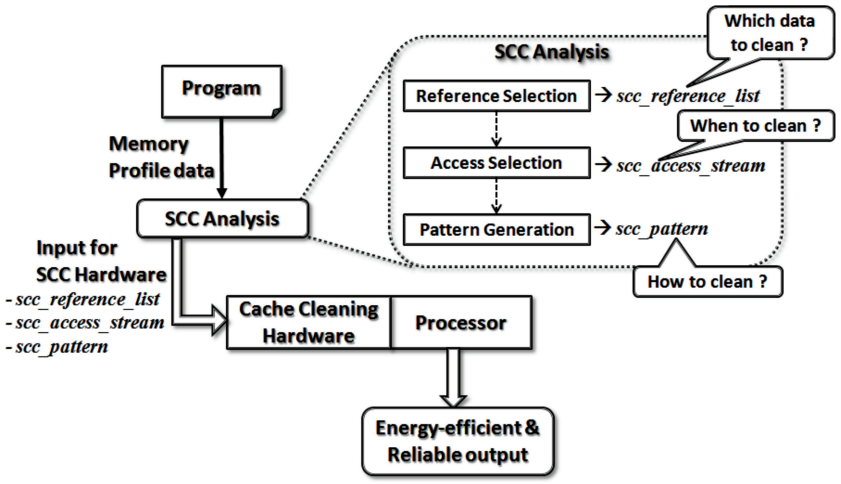


Fig. 5. Our Smart Cache cleaning methodology is described as an overview diagram representing its place in the larger scheme of things. The dotted boxes indicate internal specifics involved in the *scc analysis*. The questions each stage answers, in the implementation of SCC, is also indicated.

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 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 Figure 5) 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.

In effect 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. Our synthesis results on a RTL implementation of the SCC hardware architecture (as shown in Figure 4) indicate that the additional hardware incurs an 1.2% are and 0.06% power overheads over that of a unmodified processor core. We observe that the targeted cache cleaning hardware exists in most modern embedded processors in the form of cache-flush execution units, for for instance, the `CLFLUSH` instruction in x86 processor architectures; `MCR` (Clean and flush the line); and `cacheflush` instructions in the ARM,

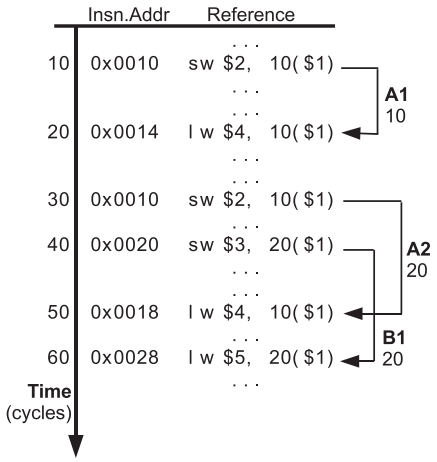


Fig. 6. Demonstrating Smart Reference Selection: On the memory profile of a program over its execution time-line, arrays A and B are accessed by instruction addresses 0x0010 and 0x0020 respectively. The individual IVT values (A1, A2, B1) are labeled and annotated by arrows that connect their W and R accesses points.

StrongARM, XScale processor architectures [ARM 2007]. For the implementation of our SCC technique with such processor architectures, the cache-line copied to the memory should not be invalidated from the cache, thereby allowing for further cache accesses on non-dirty cache-lines. These architecture hardware structures 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 Figure 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 nonoverlapping execution timelines.

The program in Figure 6 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

Table I. Data Table

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

Data table derived for statistics on the example program in Figure 6.

in Figure 6 we derive the data table Table I. 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 nonoverlapping execution timelines condition.

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

The bit stream (*scc_access_stream* denoted by the list <csw_list>) 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_Generation* algorithm to analyze the bit stream and derive a representative k bit pattern.

The *SCC_Bit_Pattern_Generation* 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

ALGORITHM 1: SCC_Pattern_Generation ()

Require: scc_access_stream <csw_list>, scc_clean_pattern_size K.

```
1: for k from 0 to K do
2:    k_ones ← Count 1^s in csw_list
3:    k_zeros ← Count 0^s in csw_list
4:    Cost_of_0 ← k_ones × 2
5:    Cost_of_1 ← k_zeros × 1
6:    if Cost_o f_1 ≤ Cost_o f_0 then
7:       BitPattern[k] = 1
8:    else
9:       BitPattern[k] = 0
10:   end if
11: end for
12: return BitPattern
```

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 previous 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 time-lines. 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 [Burger and Austin 1997] `sim-outorder` cycle-accurate simulator is configured to model the Intel XScale [Intel Corporation 2000] 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 [Shrivastava et al. 2005] 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* [Anderson 1999] and *LiverMore Loops* [McMahon 1993], which are scientific, data-intensive and computation-intensive benchmarks representative of applications executed on such embedded systems. Our justification for the use of such compute-intensive loops (with low cache activity), is to demonstrate the applicability of our SCC methodology over a worst-case environment for embedded systems (traditionally known for data-intensive applications). In addition, modern high-end HPC systems are built using embedded architecture for scalability and power-efficiency; which operate such compute intensive applications. 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, . . . , 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, . . . , 2000 cycles).

6.2. Better Energy-Vulnerability Efficiency With Smart Cache Cleaning

The graph in Figure 7 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, . . . , 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 Figure 7 we observe the following.

(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)").

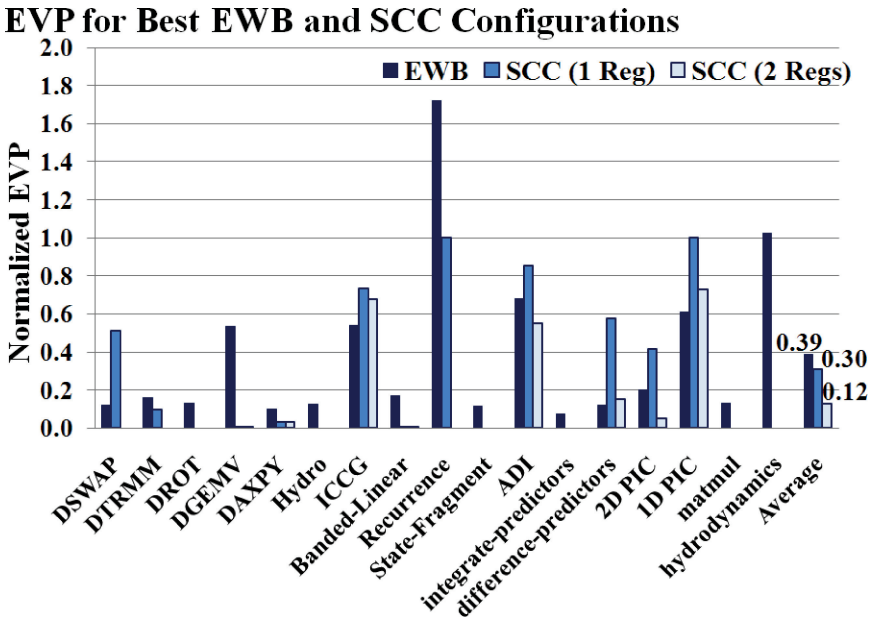
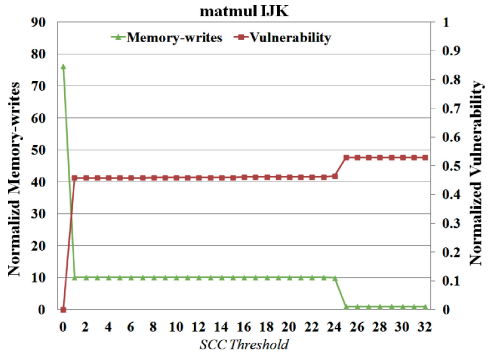


Fig. 7. The graph showing Normalized EVP of the best EWB period (EWB configuration with least EVP) and the best *scc_threshold* parameter using 1 and 2 *scc_insn_reg* registers, demonstrates higher energy-vulnerability efficiency (lower bar is better) with the SCC technique.

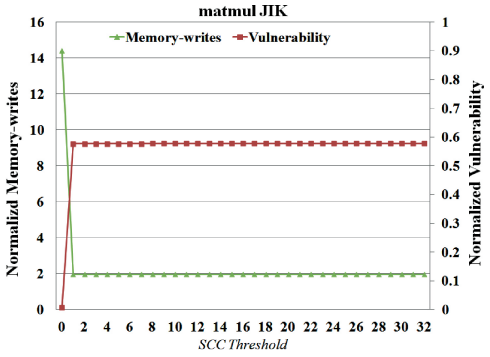
(3) From the results for the *Recurrence* benchmark, we see that the early writeback mode of cache cleaning loses on EVP by 60% compared to the original program. Owing to the complex data access pattern in the program, the best early write-back configuration (EWB period = 1800 cycles) achieves only 26% 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. Design Space Exploration

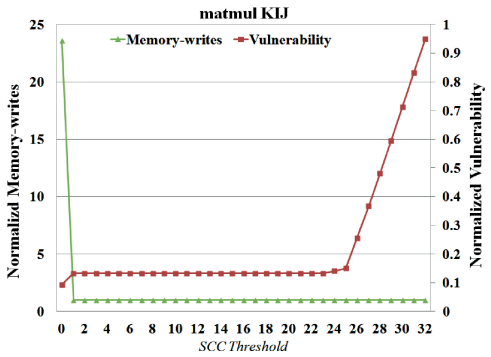
6.3.1. Choosing the Right SCC-Threshold Value. The parameter *scc_threshold* is analogous to the period of early write-backs triggered by the hardware only EWB technique. However, in this case, the parameter is associated not with the lifetime of the "application," but the vulnerable lifetime of the data after its access in the cache. In our *SCC_Analysis*, we compare this threshold value against the IVT parameter extracted using the memory profile of the application. By increasing this threshold parameter, we allow for vulnerable data to remain for longer time within the cache, while reducing the number of memory write-backs triggered because of a higher threshold value. To explore the impact of this threshold value on a program, we performed an experiment on the different loop orders of the *matmul* program. Figure 8 plots the variation in vulnerability and memory-writes for varying threshold values. For a threshold value of 0, the implementation resembles a write-through cache, and for higher threshold values we



(a) Loop order IJK



(b) Loop order JIK



(c) Loop order KIJ

Fig. 8. Impact on the number of memory write-backs and vulnerability, while varying the *scc_threshold* values for the matrix multiplication program on the L1 data cache. [In the graph, points closer to the origin are better].

observe trade-offs in vulnerability for reduced memory-writes. The best *scc_threshold* value for an application and processor configuration is that which achieves least vulnerability possible for least energy overhead.

6.3.2. SCC Across Cache Sizes. A key factor to be considered while performing profile based analysis on the memory accesses of a program, is the cache configurations. We performed experiments on varying cache sizes and configurations to determine the impact of cache size on our analysis. Though a *scc_pattern* generated is specific to the given processor configuration, this experiment demonstrates the applicability of the analysis and the SCC methodology for varying processor configurations. For increased cache-sizes, the memory profile data, and the *scc_pattern* data gathered increase proportionately. However, we note here that the *scc_pattern_generation* algorithm runtime does not depend on the size of the profile data or the *scc_pattern* data, but only on the size of the bit-pattern generated. Figure 9 plots the normalized energy and vulnerability of four benchmarks for different cache sizes, while comparing the SCC with that of the EWB hardware-only technique for cache cleaning. In the top two graphs we see clear Pareto-optimal points for all cache sizes for our SCC technique, while for the bottom two benchmarks, we see design points with no clear optimal point, and therefore trade-off analysis has to be considered in such cases. In all cases, we do observe that the change in cache configuration does not hinder the analysis, and the design points are similar for varying cache sizes for the same benchmark.

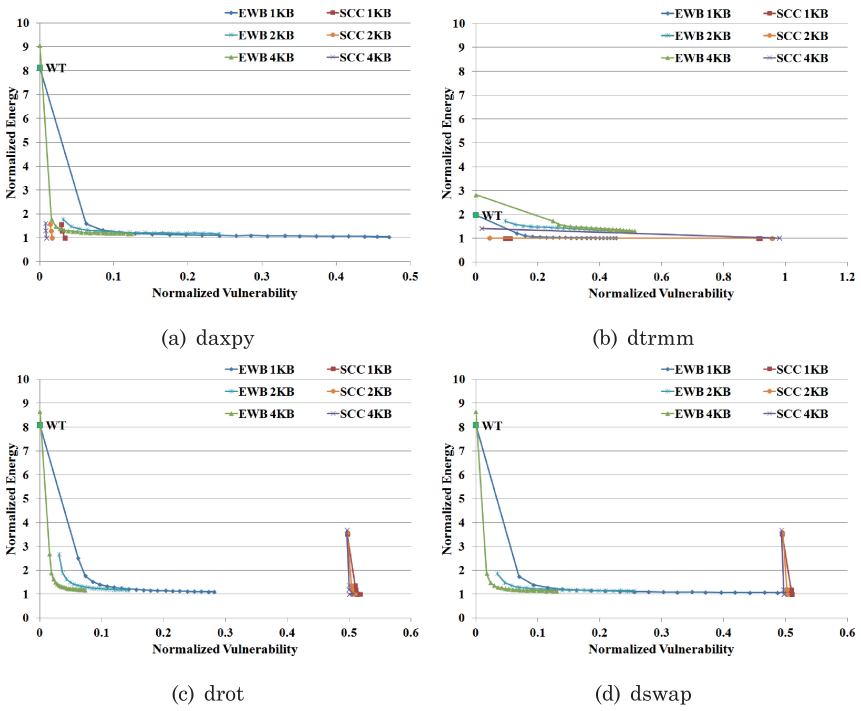


Fig. 9. Graph shows Normalized vulnerability and energy of benchmarks across different cache sizes. For the EWB mode the plot is across varying periods (100–2000), and for the SCC mode the *scc_threshold* parameter is varied a fixed *k*-bit pattern size of 32. The design points of the SCC technique can be seen closer to the axis (with minimum energy consumption and vulnerability) across cache sizes. [In the graph, points closer to the origin are better].

6.4. More Energy-Efficient Design Points in SCC

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 Figure 10, 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, . . . , 2000) and the normalized vulnerability and energy overhead incurred are plotted in Figure 10 labeled EWB. Similarly, for the same benchmark the vulnerability and overhead values are plotted for varying threshold values (5, 10, 15, . . . , 200), using our SCC technique. In the graph plotted in Figure 10, 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.

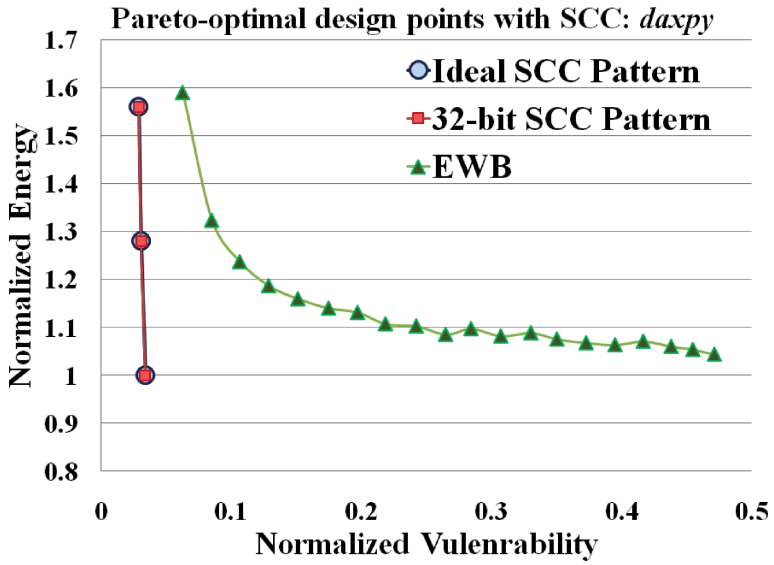


Fig. 10. Normalized vulnerability and energy plots for two benchmarks across varying *SCC_Threshold* values. The plots for *32-bit Pattern SCC* closely follow that by *Ideal SCC* in *DSWAP*, while they overlap in DAXPY, demonstrating the accuracy of Algorithm 1. Vulnerability and energy trade-off is observed for varying threshold values for each benchmark. [In the graph, points closer to the origin are better].

6.5. 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. Figure 11 plots the EVP on a log scale for the different *SCC_Pattern* sizes, across benchmarks. We observe here that for the set of benchmarks which are rather regular loops, the variation across sizes is not significant. However, in the case of `dswap`, `drot` and `hydro-dn` benchmarks, we see significant variation and therefore this parameter has to be considered as a design parameter along with the hardware overhead incurred in increasing bit-sizes. In Figure 10, 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 Figure 10, the values obtained after pattern matching on a 32-bit register closely follow that of the ideal access stream (*CSW Access Stream*). Figure 12 shows a similar plot for `hydro` and `dgemv` benchmarks. The amplified plot demonstrates clearly the accuracy of our pattern generation algorithm. 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.

6.6. 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. Figure 13 plots the EVP values of four benchmarks for varying numbers of references selected to be cleaned simultaneously. For each benchmark, the maximum

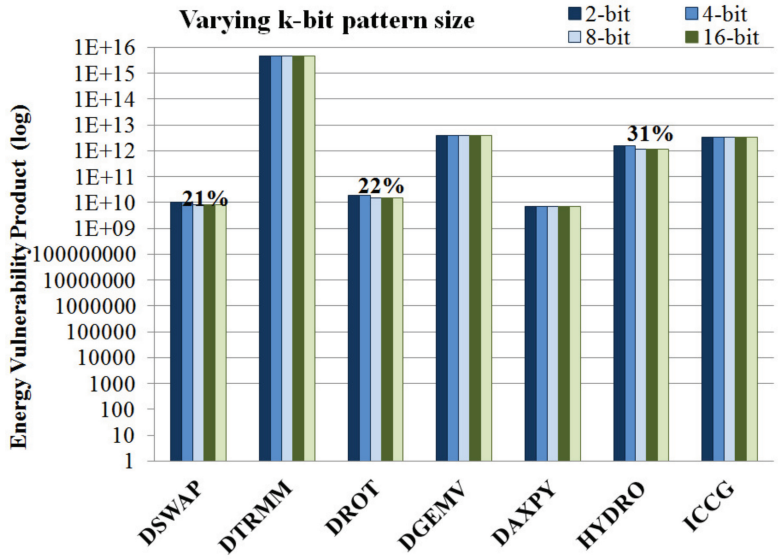
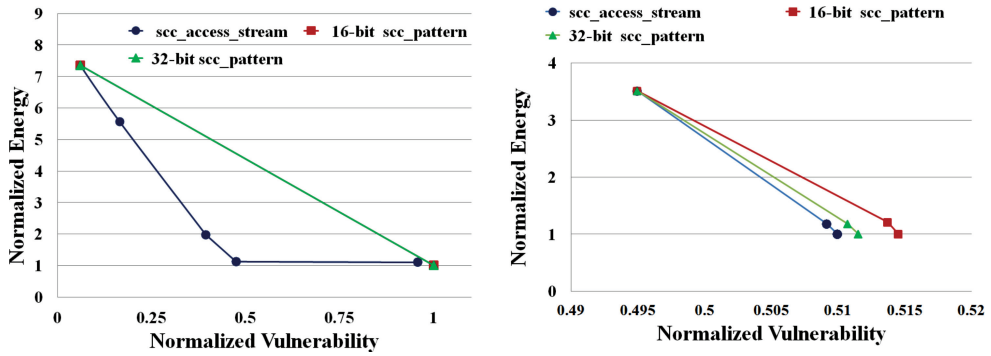


Fig. 11. Graph shows Normalized EVP of the best *scc_threshold* parameter for each of the different *k*-bit pattern sizes chosen for *scc_pattern* generation. EVP is plotted in logarithmic scale. For a benchmark, the variation in the EVP across pattern sizes is labeled on top of the group of bars. In the other cases, no discernible variation in EVP was recorded.



(a) hydro-dn (b) dgemv

Fig. 12. Graphs plotting normalized energy and vulnerability using two different sizes of k-bit *scc_pattern*, with that of the complete *scc_access_stream* from the *SCC_Analysis*. The pattern generated using our *SCC_Pattern_Matching* algorithm, gives an accurate representation of the *scc_access_stream* as a k-bit pattern.

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.

Increase in the number of references selected to be cleaned simultaneously maps to an increase in the number of *scc_insn_reg* and *scc_pattern* registers in the cache cleaning hardware. In the case that greater number of such registers are allowed in

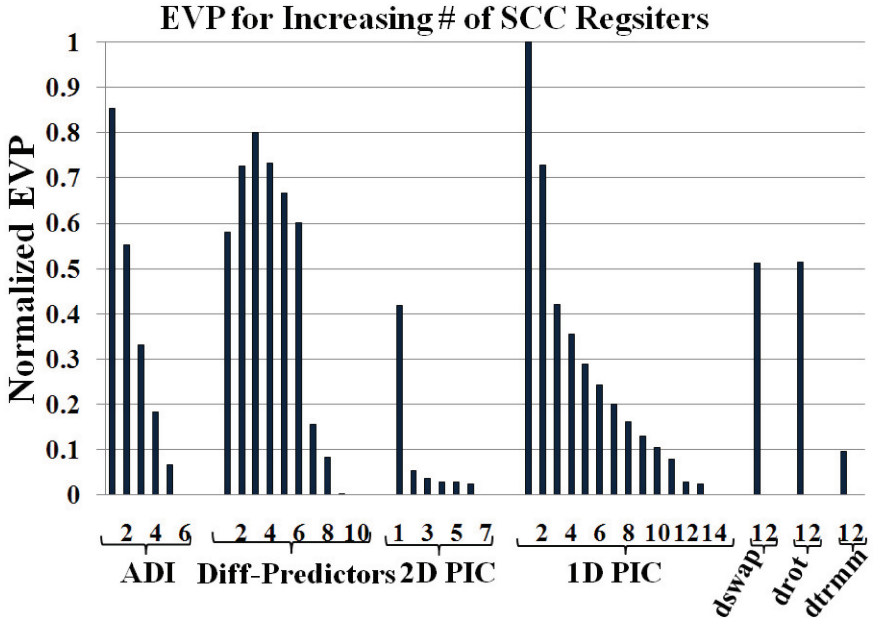


Fig. 13. The EVP of the application reduces with increase in the number of *scc_insn_reg* registers used. [In graph, lower bars are better].

the system, our SCC technique provides for scalability and thereby more energy efficient vulnerability reduction in the system. Figure 13 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 is given by that determined through memory analysis. We observe here that, in each benchmark the small additional hardware registers translate into significant EVP reduction and thereby efficient reliability of the system. It is interesting to note here that in the case of *Diff-Predictors* benchmark, with the choice of 2 and 3 registers the greedy nature of selecting references to clean simultaneously translates into bad EVP results, but when the number increases to 7, the EVP reduction is significant. The metric used to sort each reference is nothing but the vulnerability per cache write-back ratio, which from this example shows a degree of inaccuracy. A more intelligent metric and selection process would guarantee efficient choices in parameter values. We intend to pursue this direction in our future work.

7. SOFTWARE-ONLY SMART CACHE CLEANING

7.1. Methodology Overview

Keeping with the primary objective of this work, we recognize the hardware area and power overheads, though miniscule, incurred by the *Cache Cleaning Hardware* (Figure 5) required. Though the added hardware is relatively insignificant, the accesses of these hardware in conjunction with each cache access indicates additional unnecessary power overheads. In this section, we propose to utilize existing cache flush hardware architectures and their corresponding instruction triggers, to implement a software-only smart cache cleaning technique. Figure 14 describes our software-only methodology. We observe that most modern processors contain targeted cache-line write-back functionality and specific instructions to trigger them. For instance, the x86 architectures provide for the `CLFLUSH` instruction in their ISA. The `MCR` (Clean and flush

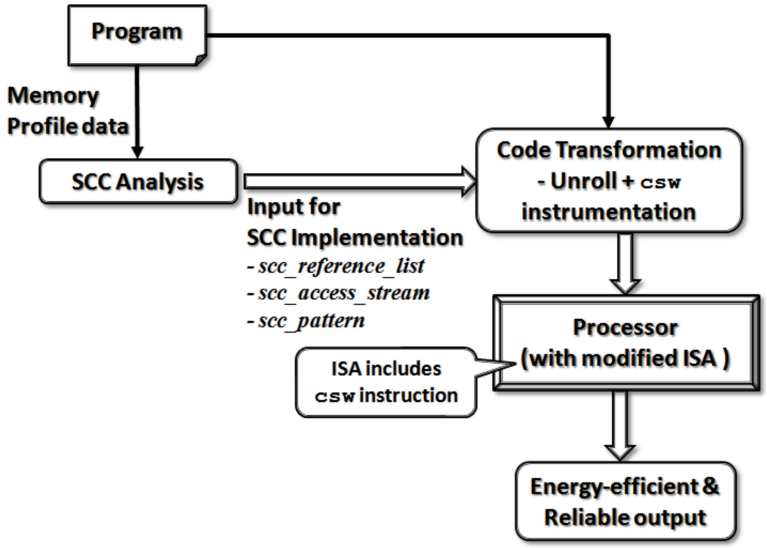


Fig. 14. The implementation of the software-only SCC technique is described.

the line) and `cacheflush` instructions in the ARM, StrongARM, XScale processor architectures [ARM 2007] provide a similar interface for embedded processors. The function of these instructions is to target either all dirty cache-blocks or specified cache-lines, as the interface may be, and write a copy of the same into the memory. Therefore, we indicate that our proposed methodology is a power-efficient software-only scheme, with negligible hardware overheads (if such hardware has to be implemented). Our software-only compiler solution achieves an additional 28% performance improvement, for the same reliability as that of our hybrid technique. It should be noted here that, the performance improvement is achieved as a result of loop unrolling code transformation on the code.

7.2. Experimental Evaluation

In order to eliminate the need for the SCC hardware to load the bit pattern, and then iterate over each bit for each cache access by the specified `store` instruction, in this work, we propose a pure software-only solution to achieve the same effects using the cache-flush instruction interfaces available. For simplicity, we consider a common instruction interface for this functionality- `csw`. The given loop can be unrolled and thus we have k instances of the specified `store` instruction as chosen from the *SCC_Analysis*. Among the k instances, the derived *scc_pattern* determines the instances that have to be cleaned, and those that are not to be cleaned. The instances that need to be cleaned can be replaced by a new instruction `csw`- clean and store word. In this, no significant decode or control logic needs to be added for its implementation. In addition to performing the `store` operation, the cache-flush logic within the processor is triggered for the targeted cache block. This performs the required cache cleaning on the specified instance on the specified cache block. Our RTL synthesis experiments implementing this targeted cache write-back architecture on a MIPS processor, incurs only around 0.56% area and negligible power overhead.

Figure 15 plots the experimental results as a proof of concept that our software-only solution achieves the same benefits as that of the hybrid SCC technique. In this, we

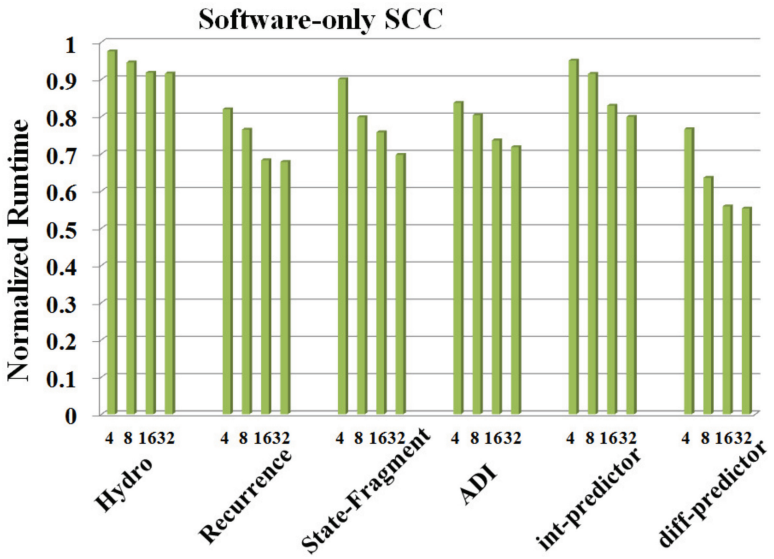


Fig. 15. The EVP of the application using SCC implementation on varying orders of unrolled loops, which correspond to the different k-bit pattern sizes identified through the *SCC_Analysis* performed. [In graph, lower bars are better].

also see instances where the performance of the loop can be improved owing to the instruction scheduling on a superscalar processor made possible by the loop unrolling code transformation. In all the cases, the energy-vulnerability product achieved was minimum as that indicated in our previous experiments. We note here that through the our software-only smart cache cleaning methodology, we achieve negligible data cache vulnerability, and minimal energy overhead; while requiring negligible hardware overheads.

8. CONCLUSION

In the race for developing smaller and smarter devices, todays and future embedded systems are susceptible to radiation induced transient errors called soft errors, even at the sea level. The cache, occupying the majority of chip area and with its unique architecture and circuitry is the most susceptible architecture component in the processor. 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. We then propose a software-only compiler technique that achieves the same energy-efficient reliability of the system, in conjunction with the code transformations implemented.

9. 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.

REFERENCES

AMD Corporation. 2007a. AMD Athlon Processor Product Data Sheet. support.amd.com/us/Processorn TechDocs/43042.pdf.

Anderson, E. 1999. *LAPACK Users' Guide*. Vol. 9, Siam, Philadelphia, PA.

ARM. 2007. ARMv5 Architecture Reference Manual. (2007). infocenter.arm.com.

Baumann, R., Hossain, T., Murata, S., and Kitagawa., H. 1995. Boron compounds as a dominant source of alpha particles in semiconductor devices. In *Proceedings of the 33rd Annual IEEE International Reliability Physics Symposium*. IEEE, 297–302. DOI: http://dx.doi.org/10.1109/RELPHY.1995.513695.

Burger, D. and Austin, T. M. 1997. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News 25*, 3, 13–25. DOI: http://dx.doi.org/10.1145/268806.268810.

Cannon, E. H., Reinhardt, D. D., Gordon, M. S., and Makowenskyj, P. S. 2004. SRAM SER in 90, 130 and 180 nm bulk and SOI technologies. In *Proceedings of the 42nd Annual Reliability Physics Symposium*. IEEE, 300–304. DOI: http://dx.doi.org/10.1109/RELPHY.2004.1315341.

Chen, G., Kandemir, M., Irwin, M. J., and Memik, G. 2005. Compiler-directed selective data protection against soft errors. In *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'05)*. ACM Press, New York, 713–716. DOI: http://dx.doi.org/10.1145/1120725.1121000.

Hamming, R. W. 1950. Error detecting and error correcting codes. *Bell System Tech. J*. 29, 2, 147–160.

Hareland, S., Maiz, J., Alavi, M., Mistry, K., Walsta, S., and Dai, C. 2001. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *Proceedings of the Symposium on VLSI Technology*. IEEE, 73–74. DOI: http://dx.doi.org/10.1109/VLSIT.2001.934953.

Hung, L. D., Irie, H., Goshima, M., and Sakai, S. 2007. Utilization of SECDED for soft error and variation-induced defect tolerance in caches. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'07)*. ACM, 1–6. DOI: http://dx.doi.org/10.1109/DATE.2007.364447.

Ibe, E., Taniguchi, H., Yahagi, Y., Shimbo, K.-I., and Toba, T. 2010. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *IEEE Trans. Electron Devices 57*, 7, 1527–1538. DOI: http://dx.doi.org/10.1109/TED.2010.2047907.

Intel Corporation. 2000. Intel XScale technology overview. intel.com/design/intelxscale.

Intel Corporation. 2007b. Intel IA-32 Developer's Manuals. intel.com/products/processor/manuals/.

Kayali, S. 2000. Reliability considerations for advanced microelectronics. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC'00)*. IEEE, 99–. http://portal.acm.org/citation.cfm?id=826038.826937.

Lee, K., Shrivastava, A., Dutt, N., and Venkatasubramanian, N. 2010. Partitioning techniques for partially protected caches in resource-constrained embedded systems. *ACM Trans. Des. Autom. Electron. Syst. 15*, 4, Article 30. DOI: http://dx.doi.org/10.1145/1835420.1835423.

Lee, K., Shrivastava, A., Issenin, I., Dutt, N., and Venkatasubramanian, N. 2009. Partially protected caches. *IEEE Trans. VLSI Syst. 17*, 9, 1343–1347. DOI: http://dx.doi.org/10.1109/TVLSI.2008.2002427.

Li, J.-F., and Huang, Y.-J. 2005. An error detection and correction scheme for RAMs with partial-write function. In *Proceedings of the IEEE International Workshop on Memory Technology, Design, and Testing (MTDT '05)*. IEEE, 115–120. DOI: http://dx.doi.org/10.1109/MTDT.2005.16.

Li, L., Degalahal, V., Vijaykrishnan, N., Kandemir, M., and Irwin, M. J. 2004. Soft error and energy consumption interactions: a data cache perspective. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '04)*. IEEE, 132–137. DOI: http://dx.doi.org/10.1109/LPE.2004.1349323.

May T. C., and Woods, M. H. 1979. Alpha-particle-induced soft errors in dynamic memories. *IEEE Trans. Electron Devices 26*, 1, 2–9. DOI:http://dx.doi.org/10.1109/T-ED.1979.19370.

McMahon, F. H. 1993. L. L. N. L. Fortran Kernels Test: MFLOPS. www.netlib.org/benchmark/livermorec.

Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T. 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*. ACM, 29–40. DOI: http://dx.doi.org/10.1109/MICRO.2003.1253181.

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

Naseer, R., Boulghassoul, Y., Draper, J., Dasgupta, S., and Witulski, A. 2007. Critical charge characterization for soft error rate modeling in 90nm SRAM. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE, 1879–1882. DOI: http://dx.doi.org/10.1109/ISCAS.2007.378282.

Rockett Jr, L. R. 1992. Simulated SEU hardened scaled CMOS SRAM cell design using gated resistors. *IEEE Trans. Nucl. Sci. 39*, 5, 1532–1541. DOI:http://dx.doi.org/10.1109/23.173239.

Shrivastava, A., Issenin, I., and Dutt, N. 2005. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. ACM, New York, 90–96. DOI: http://dx.doi.org/10.1145/1086297.1086310.

Shrivastava, A., Lee, J., and Jeyapaul, R. 2010. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'10)*. ACM, New York, 143–152. DOI: http://dx.doi.org/10.1145/1755888.1755910.

Slayman, C. 2010. Alpha Particle or Neutron SER-What will dominate in future IC technology. ewh.ieee.org/soc/cpmt/presentations/cpmt0910e.pdf.

Sridharan, V., Asadi, H., Tahoori, M. B., and Kaeli, D. 2006. Reducing data cache susceptibility to soft errors. *IEEE Trans. Dependable Secure Comput. 3*, 4, 353–364. DOI: http://dx.doi.org/10.1109/TDSC.2006.55.

Zhang, W. 2009. Computing and minimizing cache vulnerability to transient errors. *IEEE Des. Test Comput. 26*, 2, 44–51. DOI: http://dx.doi.org/10.1109/MDT.2009.29.

Received April 2012; revised August 2012, February 2013; accepted March 2013