Cache Vulnerability Equations for Protecting Data in Embedded Processor Caches from Soft Errors

Aviral Shrivastava
Compiler and Microarchitecture Laboratory, Arizona State University, Tempe, AZ 85281, USA
aviral.shrivastava@asu.edu

Jongeun Lee
High Performance Computing Laboratory, Ulsan National Institute of Science and Technology, Ulsan, South Korea
jlee@unist.ac.kr

Reiley Jeyapaul
Compiler and Microarchitecture Laboratory, Arizona State University, Tempe, AZ 85281, USA
reiley.jeyapaul@asu.edu

Abstract

Continuous technology scaling has brought us to a point, where transistors have become extremely susceptible to cosmic radiation strikes, or soft errors. Inside the processor, caches are most vulnerable to soft errors, and techniques at various levels of design abstraction, e.g., fabrication, gate design, circuit design, and microarchitecture-level, have been developed to protect data in caches. However, no work has been done to investigate the effect of code transformations on the vulnerability of data in caches. Data is vulnerable to soft errors in the cache only if it will be read by the processor, and not if it will be overwritten. Since code transformations can change the read-write pattern of program variables, they significantly effect the soft error vulnerability of program variables in the cache. We observe that often opportunity exists to significantly reduce the soft error vulnerability of cache data by trading-off a little performance. However, even if one wanted to exploit this trade-off, it is difficult, since there are no efficient techniques to estimate vulnerability of data in caches. To this end, this paper develops efficient static analysis method to estimate program vulnerability in caches, which enables the compiler to exploit the performance-vulnerability trade-offs in applications. Finally, as compared to simulation based estimation, static analysis techniques provide the insights into vulnerability calculations that provide some simple schemes to reduce program vulnerability.

Categories and Subject Descriptors C.4 [*PERFORMANCE OF SYSTEMS*]: Fault tolerance; C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems

General Terms Design, Measurement, Reliability, Theory

Keywords cache vulnerability, static analysis, soft errors, code transformation, compiler technique, embedded processors

1. Introduction

Soft errors are becoming an ever important concern for electronic system designs manufactured in deep sub-micrometer fabrication technologies. Soft errors are transient faults caused due to several sources, such as static noise in digital circuits, interconnect coupling, charge sharing noise etc., but radiation particle strikes are responsible for more transient faults than all the other causes combined [30]. Soft errors are especially important for embedded systems which may be used inside humans, close to humans, in financial, medical, and security transactions, and even in hostile and enemy territory, where there is a critical need for dependable information. Although the soft error rate in embedded devices such as handhelds is about once-per-year today, due to its exponential growth rate with technology generations, it is expected to reach alarming levels of once-per-day in about a decade [13].

Inside a processor, memory elements are most susceptible to soft errors, not only because they are typically the largest structures by area and transistor count, but also because there is no logical and temporal masking of soft errors in memories, and they operate on lower voltage swings [5, 9, 17, 31]. In fact, according to [18], more than 50% of soft errors happen in memories. Lower levels of memories (farther from processor) can be relatively easily protected using ECC (Error Correcting Code) based techniques, but protecting memories closer to the processor (i.e., L1 caches) results in high overheads. Previous research [16, 23] has shown that implementing SEC-DED (Singe-Error Correction and Double-Error Detection) can increase L1 cache access latency by up to 95%, power consumption by up to 22%, and area cost by up to 18%. Even if the performance overhead could be hidden, the power and area overheads cannot. Moreover, due to high degree of process variations, SEC-DED in caches is increasingly being used in covering up for manufacturing defects, leaving only parity checking for many cache blocks. The other option of implementing double-bit error correction has extremely high overheads [1, 21]. Another popular approach is to use write-through L1 caches. Write-through L1 caches ensure at least two copies of the latest data, therefore, they drastically reduce the vulnerability of data in caches, but it greatly increases the memory traffic between the processor and the lower levels of memory. Consequently, they are not desirable for multi-core and multi-processor systems [12].

This paper explores an orthogonal solution space for protecting data in caches - through software techniques. We observe that data is vulnerable to soft errors in the cache only if it will be read by the processor, and not if it will be overwritten. Basic code transformations like loop interchange, loop fusion, and data layout transformations like array interleaving, and array placement can change the read/write pattern of variables in the cache, and therefore should have significant effect on the vulnerability of data in the cache.

LCTES'10, April 13–15, 2010, Stockholm, Sweden.

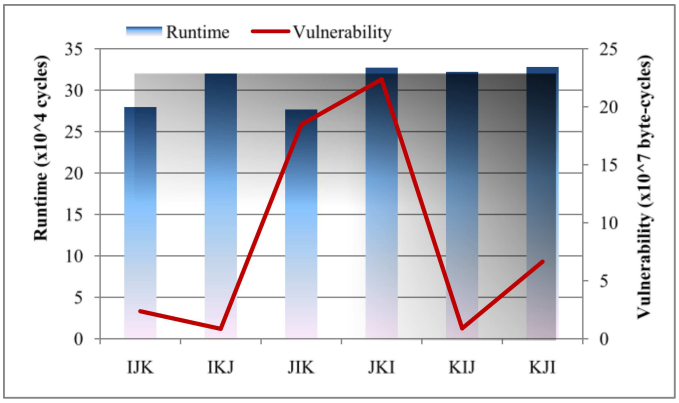

Figure 1. Runtime and data cache vulnerability for different loop orders of matrix multiplication. **Observation 1:** Variation in vulnerability = 96%, Variation in runtime = 16%. **Observation 2:** Loop order IJK has low runtime and low vulnerability. **Observation 3:** Runtime and vulnerability do not follow the same trend.

In order to demonstrate the effect of code transformations on vulnerability and to motivate for the need for techniques to analytically estimate vulnerability, we perform a simple experiment of loop interchange on the matrix multiplication kernel. The application involves three 2D arrays of 32x32 words, and data cache of 1KB size, 32-byte block, direct-mapped, with write-back, and write-allocate policies. L1 data cache vulnerability is measured using cycle-accurate simulation. Figure 1 shows the vulnerability and runtime results for all six loop orders.

The **first observation** from the graph is that there is a much greater variation in vulnerability (96%, from JKI to IKJ) than in runtime (16%, from KJI to JIK). This shows that there is an interesting trade-off between vulnerability and runtime – in the sense that vulnerability can be significantly reduced at low runtime overhead.

The **second observation** we make from this graph is that IJK loop order has low runtime, and low vulnerability. In fact as compared to the least runtime loop order, JIK, we increase runtime by less than 1%, while reducing the data-cache vulnerability by more than 4X. This motivates for the need of finding such design/execution points, which simultaneously optimize runtime and vulnerability.

Our **final observation** from the graph is that the trend of runtime and vulnerability is not dependent, and cannot be derived from one another. This is a little counter-intuitive since to a first order of approximation, increase in runtime should imply an increase in the vulnerability, since the data spends more time in the cache. However, vulnerability depends on many other factors including program's data access pattern, cache parameters, and data placement.

Therefore, to be able to find design/execution points which are good both in terms of runtime and vulnerability, we need a scheme to estimate the vulnerability of data in caches. Only cycle-accurate simulation based techniques are known to estimate cache vulnerability. While they can certainly be used (e.g. in our motivating example) to explore some code transformations, and optimize for vulnerability and runtime, however there are limitations. Cycle-accurate simulation is very slow (only a few Kilo instructions-per-second [4]), and the design space for some compiler transformations can be very large. For example, using cycle-accurate simulation to explore design space for array placement for even a 32 x 32 matrix multiplication will take months on a 2 GHz dual-core processor system. Thus there is a need for efficient techniques to estimate cache vulnerability of programs.

To this end, in this paper, we develop analytical techniques to estimate data vulnerability in caches. In spite of analytic techniques being efficient, they provide insights which can be used to develop simpler techniques to approximate; which is another critical limitation of simulation based techniques.

This paper makes several key contributions:

- Demonstrate that code transformations can significantly affect vulnerability of program loops. Often, it is possible to trade-off little performance loss for significant vulnerability reduction.
- Develop static analysis to accurately estimate vulnerability of affine loops. This includes making suitable approximations to trade-off accuracy of vulnerability estimation to computational complexity.
- Realizing that vulnerability estimation is complex, we show how our understanding from analytical vulnerability calculations can be used much more simply.

2. Related Work

Solutions to mitigate the impact of soft errors are being sought after at all levels of computer design e.g., careful selection and screening of materials [2], SOI fabrication technologies [6], increasing the transistor size, adding passive capacitance, or changing the transistor types with threshold voltage shifts, adding gated resistors [29], partially protected caches [15], software duplication [28], to triple modular redundancy [25].

As opposed to hardware techniques, software techniques reserve the advantages of *flexibility* of application, and therefore the overheads thereof. Indeed, the most important benefit of software schemes is as a last-minute fix. For example, if it is required to improve the system reliability after system design, then only software techniques may be easily applicable. Most existing software approaches that attempt to improve reliability and mitigate the effect of soft errors are based on some form of program duplication [8, 11, 22, 28], and therefore incur severe power and resource overhead. This work is fundamentally different from all those previous software approaches – we study code transformations that will improve the reliability of application programs – without re-executing any instruction of the program. Therefore our techniques can have much less power, performance, and resource overheads.

Caches are one of the most vulnerable microarchitectural components in the processor and several techniques have been developed to reduce failures due to soft errors in caches, e.g. [3, 15, 20, 32], however the effect of code transformations, and in general compilers has not been evaluated. While there has been recent work in developing compiler techniques for register file protection [14, 35], there are no compiler approaches to mitigate the impact of soft errors in caches. Vulnerability [19] is the measure of failure rate of caches, and only simulation-based techniques are known to estimate it [32]. The ability to estimate the vulnerability for any given code is fundamental to not only driving, but even developing any compilation technique to optimize for vulnerability, and in general, simulation based techniques are not usable. This underscores the need for more efficient techniques to estimate vulnerability of data in caches.

This paper proposes a static analysis to estimate program analysis, and our approach builds upon cache miss analysis [10]. While there is a more general approach [7] to model reuses in Presburger arithmetic [26], we use the reuse-vector based approach [34], since it is much more tractable. We use the Omega library [26] to perform polygon union and intersection operations and Polylib [24] to count the number of points in the polygons containing vulnerable iterations.

3. Background

3.1 Program Model

As is common with many static loop analysis techniques (e.g., [7, 10]), we consider a single loop nest, whose loop bounds and array index expressions are defined by affine functions of the enclosing loop indices. We also assume that all the load/store references inside a nest correspond to only the array references. Scalars can be analyzed as single element arrays. We use reuse vectors to find the last access, which further requires that references generate memory addresses in a uniform manner. In this paper we consider only perfectly nested loops and assume that the loop body has no conditional statement other than the loop itself. We also assume that memory accesses are made only through array references and arrays do not overlap (no alias). In practice, however, the constraints in our program model are not too restrictive. [10] showed by an empirical study, that most of the runtime-wise important loop nests in standard benchmark suits, like SpecFP are amenable to these analysis constraints.

3.2 Architecture Model

The basic architecture modeled here is a uni-processor model with a single-level, data cache hierarchy. It will be possible to extend the analysis to multi-level cache hierarchies, but, lower level of caches are relatively easily and routinely protected through ECC-based hardware techniques. We assume a direct-mapped cache with write-allocate policy, implying that if the processor writes to a cache block, and the cache block is not present in the cache, it is brought into the cache before writing on it.

3.3 Terminology

A **Reference** is a static memory read or write operation in the program whereas an *access* refers to a dynamic instance of a reference [10]. In the example illustrated in Figure 2 (a), $R_a = a[j]$, and $R_b = b[i]$ are references. For $N = 4$, each of them is invoked 16 times, and each invocation is an access.

The iterations of an n-level nested loop can be represented by an n-dimensional convex polytope $\mathcal{I} \subset \mathcal{Z}^n$ bounded by the loop bounds, called **Iteration Space** (outermost loop index being the first element in the vector). Each point in an iteration space represents an *iteration* of the loop nest. Similar to Chatterjee et al. [7] we augment each iteration with reference ID to represent *access* (IDs are given in the order in which they will be accessed in the loop of interest). We then define the **Access Space** as $\mathcal{A} = \{(\vec{j}, R) \mid \vec{j} \in \mathcal{I}, R \in \mathcal{R}\}$, where $\mathcal{R}$ is the set of IDs for all references. In the example, the access space of reference R_a is shown by the 4×4 grid of points in the light square, while that of R_b is shown by the points in the dark square in Figure 2 (b).

Like iterations, accesses are ordered. An access $(\vec{j}, R)$ precedes another access $(\vec{k}, S)$ if $(\vec{j}, R)$ is lexicographically less than $(\vec{k}, S)$, or $(\vec{j}, R) \prec (\vec{k}, S)$. [1] We use access and iteration interchangeably when considering only one reference.

The mapping from an iteration to memory address for a reference is called it's *access function*, $AF_R : \mathcal{I} \rightarrow \mathcal{Z}$, and the set of all possible memory addresses accessed is the **Memory Space** of the program. Figure 2 (c) shows the memory space of the program, where both arrays have 4 elements. Array a starts from the origin, and array b starts immediately after it ends. The access function of reference R_a is $AF_{R_a}(i, j) = S_a + j$, where $S_a = 0$ is the starting address of the array a in the memory. To model caches, we note

[1] **Note:** The lexicographical size of a vector $\vec{v}$, denoted by $||\vec{v}||$ and simply called *size*, is defined as the number of points that are lexicographically less than $\vec{v}$ in the iteration space. Greater/smaller/minimum is also in the lexicographical sense.

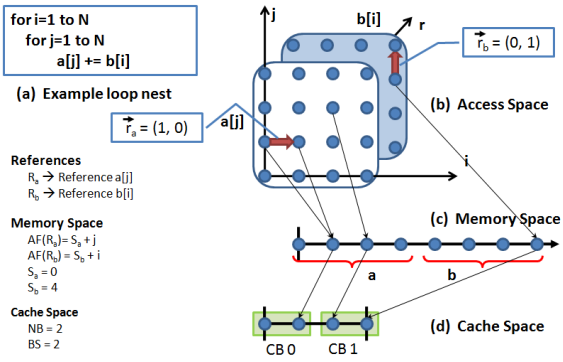

Figure 2. Access Space and Access Relations: **(a)** $R_a = a[j]$ is a reference, while each instance of it when the program executes is an access. **(b)** 16 points in the $i \times j$ space denotes the iteration space, and 2 sets of these points, one for reference R_a and one for R_b constitute the access space. **(c)** Accesses $(0, 1, R_a)$ and $(1, 1, R_a)$ have a *reuse relation* since they access the same memory address. **(d)** Accesses $(2, 2, R_a)$, and $(3, 3, R_b)$ access the same cache block, therefore they have a *conflict relation*, while accesses $(1, 1, R_a)$ and $(2, 2, R_a)$ are *unrelated* since they access different cache blocks.

that data is organized as blocks in caches. The *cache block* function CB gives the cache block number for a memory address. Thus $CB : \mathcal{Z} \rightarrow \mathcal{Z}$. The set of block numbers is called **Cache Space**. For a direct mapped cache, $CB(n) = (\frac{n}{CS})\%BS$, where CS is the cache size, and BS is the block size of the cache. In the example in Figure 2 (d), the cache has 2 blocks, each of size 2 elements.

Every pair of memory accesses have either a *reuse relation*, a *conflict relation*, or are *unrelated*. Two accesses have a **reuse relation** if they access the same memory address. In Figure 2, the access $(0, 1, R_a)$ and $(1, 1, R_a)$ access the same address in memory (corresponding to the location of $a[1]$, therefore they have a *reuse relation*. Given a memory access, $\vec{a} = (\vec{j}, R)$, any access $(\vec{k}, S)$ that has reuse relation with it is called a *reuse access* of $\vec{a}$, and $\vec{k}$ is called a *reuse iteration* of $\vec{j}$. Of particular interest is the last reuse access $\vec{a}$, and which is just the latest of reuse accesses among those that precede $\vec{a}$. Reuse vectors are used to succinctly capture last reuse access for references [34]. The reuse vector for R_a is $\vec{r_a} = (1, 0)$, and the reuse vector for reference R_b is $\vec{r_b} = (0, 1)$.

Two accesses have a **conflict relation** if they access different memory address, but the same cache block. In Figure 2, access $(2, 2, R_a)$ and $(3, 3, R_b)$ access different memory addresses, $a[2]$, and $b[3]$ respectively, but access the same cache block, $CB1$. Therefore accesses $(2, 2, R_a)$ and $(3, 3, R_b)$ conflict with each other. Finally, if both the memory addresses and the cache blocks of two accesses are different, then the two accesses are **unrelated**. In Figure 2, accesses $(1, 0, R_a)$ and $(2, 2, R_a)$ are unrelated, since they access different cache blocks.

In the absence of aliasing (e.g., the arrays are non-overlapping, the references are always accessed by their true names), there is no reuse between accesses of different references. However, there still may be conflicts between accesses of different references. Finally, if cache block size is equal to one element, there is just one reuse vector per reference to an array.

4. Cache Miss Equations

Any memory access that has a preceding reuse access but has no conflict access between itself and its last reuse access must result in a cache hit. Conversely, a conflict access between the two accesses

to the same address will cause a cache miss in a direct-mapped cache. Thus in order to identify cache hits and misses, we need to know only two things: i) last reuse access, and ii) whether a conflicting access exists or not.

Finding conflicts among accesses is easy. We know that two accesses $(\vec{j}, R)$, and $(\vec{k}, S)$ conflict iff $CB(AF_R(j)) = CB(AF_S(k))$. Finding the last reuse access is a little tricky. First lets assume that there is only one reference to an array, and there is only one reuse vector per reference. Then for the reference R, we assume that $\vec{r}$ is the reuse vector, then by definition of reuse vectors, if some memory address is accessed in iteration $\vec{j}$, then it was last accessed in iteration $(\vec{j} - \vec{r})$, and both access the cache block $CB(AF_R(j))$. Now by our definition of cache miss, there will be a miss iff, some other reference, say S, accesses the same cache block in iterations $(\vec{j} - \vec{r})$ through $\vec{j}$. This is captured in the Cache Miss Equation:

$$CME_R^S(\vec{j}, \vec{k}, \vec{r}) := (CB(AF_R(\vec{j})) = CB(AF_S(\vec{k}))) \\ \wedge ((\vec{j} - \vec{r}) \prec \vec{k} \prec \vec{j}) \quad (1)$$

It states that the reference R will experience cache miss at iteration $\vec{j}$ along the reuse vector $\vec{r}$, due to another reference S in iteration $\vec{k}$, iff they access the same cache block. If the equality is satisfied for any value of $\vec{k}$, there is a cache miss at iteration $\vec{j}$. Now we can collect the iterations in which miss occurs:

$$MI_R^S(\vec{r}) = \{\vec{j} \in \mathcal{I} \mid \exists \vec{k} \in \mathcal{I}, CME_R^S(\vec{j}, \vec{k}, \vec{r})\} \quad (2)$$

MI_R^S is the set of all iterations $\vec{j}$ in which there is a cache miss for accesses of reference R due to a conflict with another reference S, along the reuse vector $\vec{r}$.

Till now we have only considered misses because of one other reference S. If there are multiple references, then there will be a miss at iteration $\vec{j}$, if there is a conflicting access due to any of the other references. Therefore,

$$MI_R(\vec{r}) = \bigcup_{S \in \mathcal{R}} MI_R^S(\vec{r}) \quad (3)$$

$MI_R(\vec{r})$ will be the set of all iterations $\vec{j}$ at which a cache miss occurs due to a conflict with *any* other reference (except another reference to the same array) by conflicting with the reuse due to reuse vector $\vec{r}$.

Now if there are multiple references to the same array, then it will result in multiple reuse vectors. There can be more than one reuse vectors, even if the cache block size is more than one element. In that case, there is spatial reuse [10]. When there are multiple reuse vectors, then, a cache miss will occur at iteration $\vec{j}$, if there is a cache miss due to the smallest reuse vector. Noting that if there is a miss due to smallest reuse vector, then even the longer reuse vectors must suffer a cache miss, we can simply use the intersection operator. Therefore,

$$MI_R = \bigcap_i MI_R(\vec{r}_i) = \bigcap_i \Big(\bigcup_{S \in \mathcal{R}} MI_R^S(\vec{r}_i) \Big) \quad (4)$$

MI_R will contain the set of all iterations $\vec{j}$ at which a cache miss occurs for accesses of reference R due to any reuse vector, and any other reference. All the misses in the loop is then just a collection of misses of each reference.

5. Cache Vulnerability and Challenges in Estimation

Cache vulnerability (CV) is defined as the number of vulnerable bits in the cache, summed over the duration of a program execution, measured in byte-cycles. A bit is vulnerable if a soft error in it can destroy *architecturally correct execution* [19] of the processor. Any bit that is going to be overwritten is not vulnerable. Any bit in the data array that is protected with parity bit is not vulnerable if the cache block is clean and the bit is going to be accessed while the block remains clean. This is because a clean block can be simply invalidated if an error is detected in it. We assume, that all lines are protected by a parity bit, and therefore clean lines are not vulnerable.

Only cycle-accurate simulation based schemes are known for cache vulnerability estimation. While simulation based techniques are time consuming, they can be used in extremely embedded applications, where neither the program flow, or the data changes. However, if the data and its size can change, then simulation based techniques are of little help. In addition, the design space of some code transformations and data layout optimizations is so large, that exhaustive simulation is infeasible. Thus, except for in extremely embedded systems, an efficient technique to estimate data cache vulnerability is needed to decide on code transformations and data layout optimizations. An analytical model to estimate cache vulnerability offer the additional advantage of insights that we gain, and can then be utilized either apply these technique more to a different architecture/data set. In addition, it also provides a systematic and more informed mechanism to trade-off accuracy for the analysis time.

We build our cache vulnerability estimation technique similar to cache miss equations, but estimating cache vulnerability is far more complicated than cache misses. Cache miss equations estimate the number of cache misses, which is a subset of the cache accesses to the same data. In comparison, cache vulnerability is the sum of "time duration" between two consecutive accesses to the same data, when the second access is a read, and the data that was accessed was dirty. The two main complications in vulnerability estimation, as compared to estimating cache misses are: i) Notion of "time" between accesses, and ii) More information about the accesses, e.g., whether the accesses is a read or write, the knowledge of whether the data was "dirty" at the time of access.

While the second problem is simpler (in theory) and can be solved by adding more detailed information about references, the first is a fundamentally challenging problem. To compute cache misses, fundamentally for every access we only define a **Boolean function** $AM: \mathcal{A} \rightarrow \{0, 1\}$ from the access space indicating whether there was a miss at the access. The misses in the program are then just specified as a subset of the access space, i.e., $Miss = \{\vec{a} \mid AM(\vec{a}) = 1, \vec{a} \in \mathcal{A}\}$. While enumerating the elements of $Miss$ is doubly exponential [7], the number of cache misses can be found in polynomial time by simply counting the number of elements in the set [10].

In contrast, to compute cache vulnerability, we need to define an **Integer function** $AV: \mathcal{A} \rightarrow \mathcal{Z}$ which captures the vulnerability of the data since it was last accessed. The program vulnerability can then be computed by adding the vulnerabilities of each access, i.e., $Vul = \sum_{\vec{a} \in \mathcal{A}} AV(\vec{a})$. One of the main challenges in computing cache misses is of converting the integer function into sets, such that the total vulnerability can be computed by finding the number of elements in a set.

Other practical challenges in vulnerability estimation is that analysis at iteration granularity is required, as compared to estimating cache misses in which analysis at cache access granularity suffices. Furthermore, since the dirty information in caches is maintained at a block level of granularity, a whole block is considered vulnerable if any single bit in it is vulnerable. This makes modeling cache vulnerability at word or byte granularity challenging. This is because, a word may be vulnerable even if there are no access to it at all – it can be vulnerable if the blocks containing them are dirty!

Finally, even if we can exactly compute CV by considering all these factors, such a model is likely to be very complicated (as we

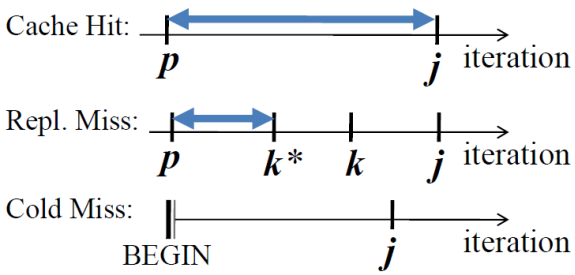

Figure 3. Access Vulnerability is the vulnerability from the last accumulated from the last access to the same data. **Cold Miss:** The vulnerable duration for the first access is 0. **Cache Hit:** The vulnerable duration is the length of the smallest reuse vector. **Cache Miss:** The vulnerable duration is the distance from the previous access to the first interfering access.

will see in the paper), so as to jeopardize its practical use. Thus, an important challenge in vulnerability estimation is also to be able to make trade-off between modeling complexity and modeling accuracy, so that we can develop a relatively simple, yet accurate model of cache vulnerability.

6. Cache Vulnerability Equations

6.1 Access Vulnerability

Unlike cache misses, which is an "event", vulnerability is computed as an "interval", or "duration". The key idea in computing vulnerability is to associate it with each access. The *Access Vulnerability*, $AV : \mathcal{A} \rightarrow \mathcal{Z}$ of an access $\vec{a} = (\vec{j}, R)$ is the vulnerability of the datum at the memory location $MF_R(\vec{j})$, since it was last accessed. If $\vec{a}$ is the first access to the data, then the datum is not considered vulnerable, or $AV(\vec{a}) = 0$. Similarly, if if this access is a write access, then it is not considered vulnerable. The reason is, that the datum is overwritten, and any error in it since the last access is inconsequential. Also if if the datum was not dirty at the time of access, then the datum is not considered vulnerable. This is because we assume that parity protection will detect the error, and the correct value can be read from the lower levels of memory, which we consider protected (through use of ECC or any other scheme).

The access vulnerability is non-zero only when the the access is a "read", and the datum was dirty at the time of access. However, the value of vulnerability depends on whether the access is a cache hit or a miss. If the access is a cache hit, then the datum was vulnerable for the whole duration from the last access to this access. Suppose that R is the only reference to the array, and it has only one reuse vector $\vec{r}$. Then the datum that is accessed by $\vec{a} = (\vec{j}, R)$ was last accessed by $\vec{b} = ((\vec{j} - \vec{r}), R)$. If the access $\vec{a} = (\vec{j}, R)$ is a cache hit, then the datum was vulnerable for the whole duration from $(\vec{j} - \vec{r})$ through $\vec{j}$, i.e., $AV(\vec{a}) = ||\vec{r}||$ (shown in Figure 3). However, if the access $\vec{a} = (\vec{j}, R)$ is a cache miss, then this datum was replaced by a conflicting access, say $\vec{c} = (\vec{k}, S)$, at iteration $\vec{k}$. Therefore, the datum was in the cache only from iteration $(\vec{j} - \vec{r})$ through $\vec{k}$. After this, from iteration $\vec{k}$ through $\vec{j}$, the datum was in the lower levels of memory, which we consider "protected". Therefore, $AV(\vec{a}) = ||\vec{k} - (\vec{j} - \vec{r})||$. However, even in the case when there are only two references, and only a single reuse vector per reference, the other reference may access and conflict more than once between accesses $\vec{b}$ and $\vec{a}$. In this case, we must consider only the distance from the last access $(\vec{j} - \vec{r})$ to the first access that interferes $\vec{k^*} = min(\vec{k})$. All these cases are illustrated in Figure 3.

6.2 Representing Integer Function

A central problem in CV modeling is how to represent the integer function AV. Recall that AM is a Boolean function, which can be easily represented as a set. Our solution is to augment the vector $\vec{j}$ in (2) with a scalar c, and let c take on all integer values less than the vulnerability l: $\{(\vec{j}, c) \mid 0 \leq c < l = ||\vec{k}^*|| - ||\vec{p}||, \cdots\}$. Essentially we diversify each $\vec{j}$ exactly l times, so that we can get the total vulnerability simply by counting the elements of the set. However, still it is not obvious how to express $\vec{k}^*$. Since $\vec{k}^*$ is the earliest conflict iteration we would like to say $\vec{k}^* = \min\{\vec{k}\}$, or $\vec{k}^* \leq \vec{k}, \forall \vec{k}$. However, $\vec{k}$ is already qualified with existential quantifier ($\exists$), and moreover adding universal quantifier ($\forall$) causes the equation to be only a general Presburger formula and not a simpler Diophantine equation, greatly increasing the complexity.

We resolve this problem by counting *nonvulnerability* instead, i.e., the size of a reuse vector minus the vulnerability. Thus we first calculate *vulnerability capacity*, and subtract nonvulnerability from it to compute real vulnerability.

Access vulnerability AV_r of reference r with only one reuse vector $\vec{v}$:

$$ANV_R^S(\vec{r}) = \{(\vec{j} \in \mathcal{I}, c) \mid \exists \vec{k} \in \mathcal{I}, \\ 0 \leq c < ||\vec{j}|| - ||\vec{k}||, \text{CME}_R^S(\vec{j}, \vec{k}, \vec{r})\} \quad (5)$$

$$ANV_R(\vec{r}) = \bigcup_S ANV_R^S(\vec{r}) \quad (6)$$

$$AV_R = ||\vec{r}|| \cdot |\mathcal{I}| - |ANV_R(\vec{r})| \quad (7)$$

Working with nonvulnerability also makes it easier to consider multiple references, as shown in (6). Further, read-hit iterations are automatically taken care of in this formula; for a hit iteration $\vec{j}$, (5) returns null since no $\vec{k}$ exists, and (7) returns the correct value. However, cold miss iterations should be excluded, which is not done by the formula.

6.3 Multiple Reuse Vectors

The formula (5)–(7) has two limitations: incorrect handling of cold miss iterations, and considering only a single reuse vector. Those two limitations are closely related and can be solved at once by extending the concept of reuse vector with *domain*. In our formulation as well as in CME, reuse vectors serve the purpose of limiting the search space for conflict miss to the previous m iterations, where m is given by the size of a reuse vector. Ideally, m should be given by the last reuse iteration (LRI). However, exact computation of LRI is intractable in the general case, but most often it can be found from reuse vectors. A reuse vector $\vec{r}$ of reference R is derived from

$$MF_R(\vec{j} - \vec{r}) = MF_R(\vec{j}) \quad (8)$$

which suggests $\vec{j} - \vec{r}$ is a possible LRI of $\vec{j}$. In order for that to be the case, two conditions must be met: i) $\vec{r}$ should be valid on $\vec{j}$ for (8), ii) there should be no smaller reuse vector valid on $\vec{j}$. We call the set of iterations where a reuse vector is valid for (8), the *domain* of the reuse vector. Certainly, domain can be defined for any reuse vector.

While there can be iterations that are not included in any domain (they are cold miss iterations), we can easily find the smallest reuse vector for any iteration that is included in at least one domain. Given a set of reuse vectors $\{\vec{r}_i\}$ sorted in their sizes, i.e., $\vec{r}_1 \prec \vec{r}_2 \prec \cdots$, and the corresponding set of domains $\{\mathcal{D}_i\}$, we define *differential domains*, $\{\mathcal{P}_i\}$, as follows.

$$\mathcal{P}_i = \mathcal{D}_i - \mathcal{D}_{i-1} - \cdots - \mathcal{D}_1 \quad (9)$$

Differential domain $\mathcal{P}_i$ is the set of iterations in which $\vec{r}_i$ is the smallest reuse vector. Clearly, differential domains are mutually disjoint, and do not include any cold miss iteration. Now we can easily extend (5)–(7).

$$ANV_R^S(\vec{r}_i, \mathcal{P}_i) = \{(\vec{j} \in \mathcal{P}_i, c) \mid \exists \vec{k} \in \mathcal{I}, \exists n \in \mathcal{Z}, \\ 0 \leq c < ||\vec{j}|| - ||\vec{k}||, \text{CME}_R^S(\vec{j}, \vec{k}, \vec{r}_i, n)\} \quad (10)$$

$$ANV_R(\vec{r}_i, \mathcal{P}_i) = \bigcup_S ANV_R^S(\vec{r}_i, \mathcal{P}_i) \quad (11)$$

$$AV_R = \sum_i \big(||\vec{r}_i|| \cdot |\mathcal{P}_i| - |ANV_R(\vec{r}_i, \mathcal{P}_i)|\big) \quad (12)$$

6.4 Access Type and Cache block State

So far we have considered how to model the effect of cache hit/miss on CV. Now we consider how to model the effect of read vs. write difference. AV_R in (12) is accurate for read accesses. For write accesses we need to exclude vulnerability due to hit accesses, which we call *hit-nonvulnerability*. Hit-nonvulnerability, HNV_R:

$$HNV_R = \sum_i ||\vec{r}_i|| \cdot |HI_R(\vec{r}_i, \mathcal{P}_i)| \quad (13)$$

$$HI_R(\vec{r}_i, \mathcal{P}_i) = \mathcal{P}_i - MI_R(\vec{r}_i) \quad (14)$$

where $MI_R(\vec{v}_i)$ is calculated by (2)–(3).

Modeling cache block state is less obvious than access type. Exact modeling requires looking even beyond LRI ($= \vec{j} - \vec{r}$) for any write to the same memory block, with the search space expanded up to the next conflict access. Compared to the formulation developed so far, which needs to find only one conflict iteration ($\vec{k}$), this new modeling requires two more (write reuse, the next conflict), which will greatly impact the complexity. Fortunately, for loops with uniformly-generated references we have a much simpler rule. Among uniformly-generated references we can define a total order from their leading/trailing relationship. For instance, in a i-j loop nest, $A[i][j]$ trails $A[i+2][j+3]$ at the distance of $[2, 3]$, which is in fact one of the reuse vectors of $A[i][j]$. We consider that a reference accesses only clean blocks if it does not follow a write reference. The other references are considered to access dirty blocks. We understand that this simple rule is only an approximation and is not always correct. However, it can be very easily applied and yet highly accurate if a write reference has no group reuse or the group reuse vector is small, which is the case in many loops including matrix multiplication.

6.5 Post-access Vulnerability

Our access vulnerability can account for only the portion of CV that becomes certain by the last access to each memory block. After the last access, there can be no more reuse but only zero or more conflict accesses. If a conflict access exists, the vulnerable interval extends to the first conflict access; otherwise, the vulnerable interval extends to the end of the program (provided that the block is dirty). Thus we need to find out i) the set of iterations in which the last accesses (per memory block) are made, and ii) the lengths of vulnerable intervals.

First, given a reference R, the set $\mathcal{P}_*$ of iterations for last accesses can be found from the *ranges*, $\{\mathcal{R}_i\}$, of reuse vectors, $\{\vec{r}_i\}$. Range is defined similarly to domain except that $-\vec{v}$ is replaced with $+\vec{v}$ in (8). Then it follows from the definition that $\mathcal{P}_*$ is the set of iterations that are not included in any range, or $\mathcal{P}_* = \mathcal{I} - \mathcal{R}_1 - \mathcal{R}_2 - \cdots$. Second, the vulnerable interval is either $|\mathcal{I}| - ||\vec{j}||$ or $||\vec{k}^*|| - ||\vec{j}||$, whichever is the smaller, where $\vec{k}^*$ is the earliest of, if any, future conflict iterations. Again we use nonvulnerability to find this interval. Post-access vulnerability of reference R is $PV_R = |U_R| - |PNV_R|$, where $U_R = \{(\vec{j} \in \mathcal{P}_*, c) \mid 0 \leq c < |\mathcal{I}| - ||\vec{j}||\}$ and PNV_R, the post-access nonvulnerability, is the union of all the post-access nonvulnerabilities PNV_R^S from different references S. Finally, $PNV_R^S = \{(\vec{j} \in \mathcal{P}_*, c) \mid 0 \leq c < |\mathcal{I}| - ||\vec{k}||, \exists \vec{k} \in \mathcal{I}, \exists n \in \mathcal{Z}, \text{CME'}_R^S(\vec{j}, \vec{k}, n)\}$, where CME′ is CME with $(\vec{j}, R) \prec (\vec{k}, S)$ substituting for the original range constraint.

Algorithm 1 Find access vulnerability of all references

```
1:  for all R ∈ 𝓡 that can access dirty cache blocks do
2:      integer (vulnerability of R): V_R ← 0
3:      Find all the reuse vectors r_i and their domains 𝒟_i
4:      for all i in the increasing order of ||r_i|| do
5:          Find 𝒫_i from {𝒟_i}
6:          for all S ∈ 𝓡 do
7:              Find ANV_R^S(r_i, 𝒫_i)   /* CME extended for CV */
8:          end for
9:          integer: ANV_R(i) ← |∪_S ANV_R^S(r_i, 𝒫_i)|
10:         V_R ← V_R + ||r_i|| · |𝒫_i| − ANV_R(i)
11:     end for
12:     if R is a write reference then
13:         Compute HNV_R using CME
14:         V_R ← V_R − HNV_R
15:     end if
16: end for
```

6.6 Implementation and Complexity

Algorithm 1 lists the procedure to compute access vulnerability for all references in a loop (post-access vulnerability can be computed similarly). The core of this procedure is writing extended CMEs (line 7) and counting the integer points in them (line 9). CME, extended or not, is a set of constraints that specify a polytope possibly using existential quantifier, and counting integer points in such a polytope can be done in polynomial time using the *barvinok* library [33], which is based on *PolyLib* [24]. However, further complication comes from the union operation in between (line 9), which exists in CME as well. There are several ways to handle unions. A simple method is to convert unions into intersections (intersections pose no problem) using the inclusion-exclusion property ($|A \cup B| = |A| + |B| - |A \cap B|$), which has unfortunately an exponential complexity. Another way is to use Pugh's method of converting unions into disjoint unions [27]. Counting the number of integer points is repeated for each reuse vector (line 4) and for each reference that can access dirty cache blocks (line 1). In our current implementation the complexity is dominated by the handling of union operator, and is $O(c \cdot N \cdot 2^{|\mathcal{R}|})$, where c is the average time for handling unions, and N is the total number of reuse vectors of references that can access dirty blocks.

7. Experiments

7.1 Vulnerability and Runtime Trade-off

Here we demonstrate through simulation, that interesting trade-off exists between vulnerability and runtime of applications. At the first glance, it seems that as the runtime increases, the vulnerability of the program should also increase, and therefore they are closely coupled. While this is true in general, there is a very significant impact of the access pattern on the cache behavior, significantly changing the amount of vulnerable data present in the cache, and therefore this direct coupling may not be realized. We show this by experimenting on several loop transformations. We collected important loop kernels from the SPEC 2000 and multimedia benchmark suites. We modified the SimpleScalar [4] simulator to com-

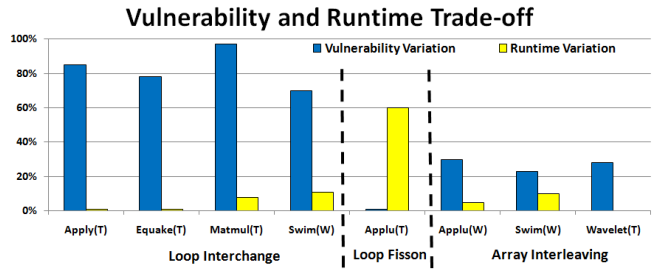

Figure 4. Runtime vs. Vulnerability: Opportunities to greatly reduce vulnerability at little performance cost exist.

pute the data cache vulnerability. The sim-outorder simulator has been parameterized to model a simple in-order embedded processor with $L1$ data cache of size $32KB$, direct mapped, with 32 byte cache block size, and 25 cycle miss penalty. The benchmarks are compiled with gcc(version 2.95.3) using the '-O' option to ensure that the compiler does reschedule the loops.

Figure 4 plots the variation in the runtime and vulnerability of the $L1$ data cache for three popular loop and data transformations, loop interchange, loop fusion, and array interleaving. For each transformation, we find the setting that results in the minimum vulnerability and the setting that results in the maximum vulnerability. The vulnerability variation is then computed as difference in the vulnerabilities of these configuration divided by the vulnerability in the maximum vulnerability setting. The runtime variation is also computed using the minimum and maximum vulnerability settings. For example, for loop interchange on matrix multiplication, Figure 1 shows, that the maximum vulnerability loop order is JKI, and the minimum vulnerability loop order is IKJ. The runtime and vulnerability variations are computed for these configurations. For loop fission there are only two setting, either the loops are fused, or they are separate (fission). Similarly there are two settings for the array interleaving case, either all the arrays are separate, or all the arrays in the loop are interleaved.

Next to application names the letter, T or W, in parentheses indicates the direction of variation; vulnerability and runtime move in the opposite directions (trade-off or T) or in the same direction (win-win or W). In about half the applications (particularly for loop interchange), we observe trade-off relationship between vulnerability and runtime, typically with much less variation in runtime (46% vulnerability variation vs. 16% runtime variation, on average). This means that for some applications we can greatly reduce vulnerability while affecting performance very little, reconfirming our motivation for cost-effective soft error approaches by compilers. In a win-win situation, on the other hand, we can get automatic vulnerability reduction by choosing performance-optimal loop transformations.

Clearly the percentage variation in runtime and vulnerability is sensitive on the relative size of the application data and cache size and other cache and memory parameters. For example, if the cache is extremely small, and all accesses miss, then will be little impact of loop orders on either cache misses, or the vulnerability. Similarly, if the cache is quite large, and there are only capacity misses, then again there will be no variation in the runtime and vulnerability of the loops. However, in general, we expect the variation in vulnerability to be much more magnified than the variation in the runtime, due to the multiplicative effect of misses in vulnerability computation. To exploit vulnerability-runtime trade-offs, techniques to estimate vulnerability are required, and efficient techniques (like Cache Miss Equations for performance) will be needed if we want the compiler to make these trade-offs automatically.

Loop order	Analytical			Simulation			
	CV(li)	#CM	ACV	CV(li)	#CM	CV(bc)	RT(c)
ikj†	2071	538	**1321**	2071	538	**1.71M**	**41.2K**
kij	5488	788	2874	5488	788	4.67M	45.9K
ijk*	6744	418	2669	6744	**418**	**5.07M**	**39.0K**
kji	15163	1746	7377	15163	1746	16.71M	68.5K
jik	33852	598	8816	33852	598	22.59M	42.5K
jki	32341	1544	11732	32341	1544	33.06M	65.4K
Corr.	1.000	1.000	.995	–			

Table 1. Vulnerability results for mmult, N=12. Legend - CV: cache vulnerability, CM: cache miss, ACV: adjusted CV, RT: runtime, (li): line-iteration, (bc): byte-cycle, and (c): cycle.

7.2 Model Validation

To validate our static analysis as well as to demonstrate its effectiveness and usefulness in program optimization we use the matrix multiply loop kernel. Our static analysis is performed using an automated analysis flow, which first derives reuse vectors and their domains from application description, then generates vulnerability equations, and finally calculates cache vulnerability using an integer-point counting engine. Simulation is performed using the SimpleScalar cycle-accurate simulator [4]. In all our experiments we assume that the L1 data cache is write-back and direct-mapped, with 32-byte line size. The cache size is set to 1∼4 KBytes depending on the application's memory footprint. Small cache sizes are chosen not only to model embedded systems but also to induce frequent cache misses, which will create more variety in the number of cache misses and cache vulnerability, and thus make it more challenging to predict the cache behavior.

Loop interchange is a well-known loop optimization that changes the order of loops in a loop nest. Since it can completely reorder the memory accesses in a loop, loop interchange can greatly affect cache vulnerability as well as cache misses. As we will see in our experimental results, there is usually much greater variation in cache vulnerability than in the number of cache misses. Moreover, the loop order with the least number of cache misses is not always the one with the lowest cache vulnerability. This suggests that in order to address reliability issues, compilers should specifically target vulnerability reduction rather than just cache miss reduction.

Table 1 compares the cache vulnerability of mmult (matrix multiplication) computed by our static analysis and by simulation. The adjusted cache vulnerability (ACV) is calculated using the number of cache misses (#CM) predicted by the CM equations [10] with modifications due to domains. The rows are sorted in the increasing order of simulation CV in byte-cycles (7th column). The last row lists the correlation coefficient between each column on the analytical side and the corresponding column on the simulation side, with ACV corresponding to CV in byte-cycles on the simulation side.

First, we can see that the second column (analytical CV in line-iterations) exactly matches the fifth column (CV in line-iteration from simulation). Mmult has nontrivial access pattern in that all three references have different pairs of spatial and temporal reuse vectors. Thus this validation result gives some assurance of our CV equations. In the table, we also observe that the number of cache misses predicted by the CM equations, with modifications due to domains, is 100% accurate as compared to simulation. Finally, the ACV numbers also closely follow the simulation results, with a very high correlation.

Besides the basic validation results, there are interesting points to observe from the table. First, the CV variation is much higher than CM variation or RT (runtime) variation. Cache vulnerability, as measured in byte-cycles from simulation, varies from 1.71M to 33.06M, or more than 19 times, whereas the number of cache misses and runtime vary by mere 3.2 and 1.7 times respectively.

Loop order	Analytical			Simulation			
	CV(li)	#CM	ACV	CV(li)	#CM	CV(bc)	RT(c)
ikj†	3622	2173	**3055**	3622	2333	**3.19M**	**92.3K**
ijk*	9892	1462	5230	10778	**1574**	**9.25M**	**79.4K**
kij	10221	2653	6520	8988	2773	10.47M	99.1K
kji	26150	3658	13575	26103	3801	35.79M	130.3K
jik	63882	1564	18549	53568	1692	49.15M	82.9K
jki	66581	3438	24793	57827	3565	81.88M	127.1K
Corr.	.9978	.9998	.9919	–			

Table 2. Vulnerability results for mmult, N=14. Legend - CV: cache vulnerability, CM: cache miss, ACV: adjusted CV, RT: runtime, (li): line-iteration, (bc): byte-cycle, and (c): cycle.

Therefore the effect of compiler optimizations for cache vulnerability can be greater than for cache misses. Second, the loop order for the minimum RT is not the same as the one for the minimum CV. The original loop order, which is marked with an asterisk in the first column, has minimum CM and consequently minimum runtime as well. However, if we choose another loop order, marked with a dagger, the cache vulnerability can be reduced by almost three times while the runtime is increased by only 5.7% (The minimum-CV loop-order can be correctly predicted by our analysis as shown in the table). Please note that the cache vulnerability in byte-cycles already takes into account the effect of increased runtime; therefore, the three times reduction in CV is the real reduction that we can expect to see in the soft error rate of the data array of L1 data cache. The above two points strongly suggest the need and scope of compiler optimizations to reduce cache vulnerability, which has been neglected in traditional loop optimizations focusing on cache misses only. Our static analysis can be an important first step toward compiler optimizations for cache reliability.

A potential weakness of our technique, as it relies on reuse vectors to simplify the equations, is the inaccuracy of reuse vectors and their domains. The prediction of reuse vectors on the last reuse access can become less accurate at the boundary of the iteration space. In our first example where $N = 12$, the iteration space is divided by the cache line size in all its dimensions (a cache line contains exactly four array elements of double word each). If we change N to 14, the boundary effect starts to appear, which is shown in Table 7.2. In the table we notice that even the CM equations start to disagree with simulation although the overall correlation is very high. Comparing Columns 2 and 5 (CV in line-iterations), our CV equations tend to be more accurate in low CV region while it amplifies in high CV region. Our CV analysis sometimes loses on the details but it accurately captures the overall trend. Most importantly, the ordering in the adjusted CV exactly matches the ordering in the simulation CV in byte-cycles. Again in this example, we observe the same pattern that the loop order for minimum CM is different from that of minimum CV, and there is much more to gain in terms of cache vulnerability if we can make a little trade-off in terms of runtime.

7.3 Analytical Optimization Case Study

Many loop transformations significantly affect cache vulnerability, often much more than cache misses. While our cache vulnerability equations can be used to accurately compute the total cache vulnerability of a loop nest, and thus can guide compiler optimizations, evaluating the equations is not always easy due to the limitations of back-end tools. Here we showcase alternative use cases of our cache vulnerability equations, using data placement.

7.3.1 Array Placement

In loops, array placement can dramatically affect the number of cache misses and cache vulnerability. There are two ways to change array placement. Intra-variable padding increases row sizes to reduce cache conflicts (both self and cross), which increases memory footprint. Inter-variable padding, or array placement adds unused space between arrays, or changes the base addresses of arrays, to reduce cache conflicts between different arrays. We use array placement to demonstrate the effectiveness of analytical optimization on cache vulnerability.

for ($i = 0$; $i < N$; i++)
  for ($j = 0$; $j < N$; j++) {
    $A_{j,i+1} = f_1(P_{j,i+1}, P_{j,i}, U_{j,i+1})$
    $B_{j+1,i} = f_2(P_{j+1,i}, P_{j,i}, V_{j+1,i})$
    $C_{j+1,i+1} = f_3(V_{j+1,i+1}, V_{j+1,i}, U_{j+1,i+1},$
      $U_{j,i+1}, P_{j,i}, P_{j,i+1}, P_{j+1,i+1}, P_{j+1,i})$
    $D_{j,i} = f_4(P_{j,i}, U_{j,i+1}, U_{j,i}, V_{j+1,i}, V_{j,i})$
  }

Figure 5. Calc1 loop from swim (after loop interchange).

Figure 5 shows an abstract version of a loop nest from swim after loop interchange (detail is omitted to avoid copyright infringement). Hereafter we refer to the loop-interchanged loop as the original loop. This loop involves 7 arrays and many more references with very complex access patterns. Exhaustive exploration of array placement parameters for such a loop is prohibitive. For a very small 1KB cache, and even after restricting the base addresses to the cache line boundary (=32B), the design space has still $(2^5)^6 = 2^{30}$ combinations. Instead, we can quickly find optimal points exploiting the intuition provided by our CV equations.

Our CV equation has two parts. It first computes the total vulnerability *capacity* and then subtracts nonvulnerability from it. Often the total vulnerability capacity is not affected by base addresses. Therefore our goal is to maximize nonvulnerability by changing base addresses. Nonvulnerability is proportional to the distance between the current iteration and the earliest conflict iteration after the previous reuse. In other words, maximum nonvulnerability occurs if a dirty cache line is evicted immediately after it is accessed, which agrees with the intuition. However, since frequent cache conflict will increase runtime, which negatively impacts cache vulnerability, our strategy is to evict as soon as possible those dirty cache lines that will *not* be accessed for a long time.

In our original loop in Figure 5, dirty cache lines are generated only by the four LHS (left-hand side) references. We can evict those lines by creating conflicts between each of them and any of the ensuing accesses. Let us use write accesses for that, since write misses generally incur less penalty. Then we have a chain of conflicts like this: $A_{j,i+1} \rightarrow B_{j+1,i} \rightarrow C_{j+1,i+1} \rightarrow D_{j,i}$. The last reference $D_{j,i}$ can be made to have conflict with one of the read references in the next iteration.

To derive formulas let us assume that all the arrays are initially placed at offset zero modulo the cache size, and that we can independently control the offset μ_X of each array X. Note that this can be implemented very easily without losing optimality. Let us also assume $\mu_A = 0$. Then the above chain of conflicts gives the offsets of the other three arrays. For instance, between A and B:

$$Addr(A_{j,i+1}) \equiv Addr(B_{j+1,i})$$
$$\Leftrightarrow \quad \mu_A + jM + (i+1) \equiv \mu_B + (j+1)M + i$$

where $\equiv$ is equality under modulo on the cache size and M is the size of each row (assuming every array has the same row size).

For the last reference, we must consider the next iteration, which can be either the next j-iteration or the next i-iteration. For each case we can set up a different array to have conflict with $D_{j,i}$. We explore three choices.

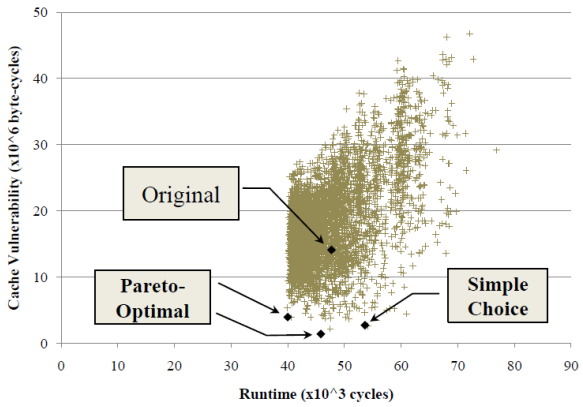

Figure 6. Cache vulnerability and runtime reduction through array placement.

Loop	CV (bc)	%reduc.	RT (c)	%incr.
Original (loop-interchanged)	14.08M	–	47.7K	–
Simple choice	2.69M	80.9%	53.6K	12.4%
Pareto-optimal 1	1.36M	90.3%	45.8K	–3.9%
Pareto-optimal 2	3.92M	72.1%	40.0K	–16.2%

Table 3. Array placement optimization results for swim

(1) Simple choice: Using $U_{j,i+1}$ and $P_{j,i}$

$$Addr(D_{j,i}) \equiv Addr(U_{j,i+1})|_{j=j+1}$$
$$\Leftrightarrow \quad \mu_D + jM + i \equiv \mu_U + (j+1)M + (i+1)$$

and

$$Addr(D_{j,i})|_{j=N-1} \equiv Addr(P_{j,i})|_{i=i+1,j=0}$$
$$\Leftrightarrow \quad \mu_D + (N-1)M + i \equiv \mu_P + (i+2)$$

Figure 6 plots CV and runtime results from simulation for random offsets (5000 instances). It also shows the CV and runtime for the original loop, which is about the center of the distribution. For parameters we use $N = 14$ and $M = 16$, and the cache size is set to 1KB. Compared to the original loop, our simple choice can reduce CV by more than 80%, which further validates our static vulnerability model. However the runtime is significantly increased. Although it is not surprising given that we have tried only to reduce CV, it suggests that cache misses should be considered in order to get truly optimal parameters.

(2) Pareto-optimal: To contain the runtime increase problem we resort to traditional CM reduction methods such as [10]. The key idea is to make the read references conflict as little as possible. Close examination of the offsets determined by the simple choice reveals that $\mu_U = \mu_C$ and $\mu_V = \mu_A$, which creates unnecessary conflicts and increases runtime. The latter is because we did not set any constraint on μ_V, which defaulted to zero, and the former is by chance. To improve the situation we set up a different reference $V_{j+1,i}$ to have conflict with $D_{j,i}$ along the j-loop, and used either $P_{j,i+1}$ or $P_{j+1,i}$ to have conflict along the i-loop, which gives two sets of parameters. Then the remaining free array U is assigned an offset that is farthest away from all the other arrays. The simulation results for these sets of parameters are shown in Figure 6 (marked as Optimal). Both points are pareto-optimal, and reduces CV by up to 90% or runtime by up to 16% compared to the original loop. Table 3 summarizes the exploration results.

8. Summary

To combat the threat of soft errors, techniques have been developed at all abstractions of processor design. Software schemes are particularly useful since they provide flexibility of application and therefore overheads, and can be applied in current or even previous generation processors, and most importantly, are irreplaceable as a last-minute fix. Caches are the most "vulnerable" component in the processor, and traditional ECC-based techniques are getting used up in manufacturing errors, and effectively only parity protection remains. While there exist some microarchitectural techniques to reduce cache vulnerability, there are no compiler based techniques. This is chiefly owing to the lack of efficient schemes to estimate cache vulnerability – for which only simulation based techniques are known. This paper develops analytical techniques to efficiently and statically estimate cache vulnerability of programs, opening the doors for compiler techniques to trade-off power and performance for reliability. Our experiments demonstrate that often it is possible to trade-off a little performance for a significant vulnerability reduction by simple code transformations. In addition, we demonstrate how the insights from vulnerability calculations can be used to innovate simple practical schemes to reduce program vulnerabilities.

Acknowledgments

This research was partially funded by grants from National Science Foundation CCF-0916652, Microsoft Research, Raytheon, SFAz and Stardust Foundation.

References

[1] A. Agarwal, B. Paul, and K. Roy. Process variation in nano-scale memories: failure analysis and process tolerant architecture. pages 353–356, Oct. 2004.

[2] R. Baumann, T. Hossain, S. Murata, and H. Kitagawa. Boron compounds as a dominant source of alpha particles in semiconductor devices. In *Anual proceedings of IEEE symposium on Reliability Physics*, pages 297–302, 1995.

[3] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-efficient soft error protection for embedded microprocessors. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 421–431, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-543-6. doi: http://doi.acm.org/10.1145/1176760.1176811.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/268806.268810.

[5] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy. Cache size selection for performance, energy and reliability of time-constrained systems. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 923–928, Piscataway, NJ, USA, 2006. IEEE Press. ISBN 0-7803-9451-8. doi: http://doi.acm.org/10.1145/1118299.1118507.

[6] 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.

[7] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. *SIGPLAN Notices*, 36(5):286–297, 2001. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/381694.378859.

[8] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 113-119, Jun 1995.

[9] J. Gaisler. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. *Fault-Tolerant Computing, Inter-*

national Symposium on, 0:42, 1997. ISSN 0731-3071. doi: http://doi.ieeecomputersociety.org/10.1109/FTCS.1997.614076.

[10] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97*, pages 317–324, 1997. ISBN 0-89791-902-5. doi: http://doi.acm.org/10.1145/263580.263657.

[11] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. *SIGARCH Comput. Archit. News*, 33(2):172–183, 2005. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1080695.1069985.

[12] L. Hung, M. Goshima, and S. Sakai. Mitigating soft errors in highly associative cache with cam-based tag. pages 342–347, Oct. 2005. doi: 10.1109/ICCD.2005.76.

[13] S. Kayali. Reliability considerations for advanced microelectronics. In *PRDC '00: Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, page 99, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0975-4.

[14] J. Lee and A. Shrivastava. Static analysis to mitigate soft errors in register files. In *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE '09.*, pages 1367–1372, April 2009.

[15] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 411–420, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6. doi: http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/1176760.1176810.

[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. doi: 10.1109/MTDT.2005.16.

[17] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 340–349, Jun 1994. doi: 10.1109/FTCS.1994.315626.

[18] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2005.70.

[19] S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, 2003. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2003.1261389.

[20] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? *Pacific Rim International Symposium on Dependable Computing, IEEE*, 0:37–42, 2004. doi: http://doi.ieeecomputersociety.org/10.1109/PRDC.2004.1276550.

[21] A. Nourivand, A. Al-Khalili, and Y. Savaria. Aggressive leakage reduction of srams using error checking and correcting (ecc) techniques. pages 426–429, Aug. 2008. doi: 10.1109/MWSCAS.2008.4616827.

[22] N. Oh, S. Mitra, and E. McCluskey. Ed4i: error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180–199, Feb 2002. ISSN 0018-9340. doi: 10.1109/12.980007.

[23] R. Phelan. Addressing soft errors in arm core-based designs. Technical report, ARM, 2003.

[24] polylib. URL `http://icps.u-strasbg.fr/polylib`. PolyLib – A library of polyhedral functions.

[25] D. K. Pradhan, editor. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-057887-8.

[26] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: http://doi.acm.org/10.1145/125826.125848.

[27] W. Pugh. Counting solutions to Presburger formulas: how and why. *SIGPLAN Notices*, 29(6):121–134, 1994. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/773473.178254.

[28] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: http://dx.doi.org/10.1109/CGO.2005.34.

[29] 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. ISSN 0018-9499. doi: 10.1109/23.173239.

[30] K. Shepard, V. Narayanan, and R. Rose. Harmony: static noise analysis of deep submicron digital integrated circuits. *IEEE Trans. on CAD*, (8):1132–1150, 1999.

[31] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. *Dependable Systems and Networks, International Conference on*, 0:389, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/DSN.2002.1028924.

[32] 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. doi: http://doi.ieeecomputersociety.org/10.1109/TDSC.2006.55.

[33] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational function. *Algorithmica*, 48(1):37–66, 2007. doi: 10.1007/s00453-006-1231-0.

[34] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113449.

[35] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *EMSOFT '05*, pages 203–209, 2005. ISBN 1-59593-091-4. doi: http://doi.acm.org/10.1145/1086228.1086266.