STATIC ANALYSIS TO MITIGATE SOFT ERROR FAILURES IN PROCESSORS

by

Reiley Jeyapaul

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2008

STATIC ANALYSIS TO MITIGATE SOFT ERROR FAILURES IN PROCESSORS

by

Reiley Jeyapaul

has been approved

November 2008

ABSTRACT

By 2011, the Integrated Circuit(IC) feature sizes are expected to be reduced to 22nm from present day 45nm, and the soft error rate will increase by 3 to 4 orders of magnitude (from one per year to one per hour). International Technology Roadmap for Semiconductors(ITRS) indicates that techniques for mitigating soft errors are crucial for future generations of Integrated Circuits. Soft errors are transient faults, mostly caused by cosmic radiations and can lead to incorrect results or total system failure. While several process technology, circuit-level and microarchitectural techniques have been proposed to combat the challenge of soft errors, efficient soft error protection can be achieved at higher levels of design abstractions. Compiler solutions to mitigate failures due to soft errors are preferred since they do not require any design modification and are applicable on existing processors. However, the primary requirement for this is a technique to statically and quantitatively estimate of the impact of soft errors on a given program and this does not exist. The only known mechanisms to quantitatively estimate the impact of soft errors (vulnerability) is through simulation. Clearly simulation based approaches to estimate vulnerability are unusable for compiler transformations. This work presents the first static analysis to quantitatively estimate the vulnerability of programs to drive compiler optimizations. The static estimates, generated by the static analysis, are accurate to within 1 percentage error, and are able to drive code transformations to achieve significant reduction in vulnerability at minimal performance degradation.

To

Kevin, my little star

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Aviral Shrivastava, but for whom this work would have been an unplanted seed. I would like to thank him for those long, energetic and stimulating discussions with the marker and wall. His infinite patience, support, invaluable guidance and encouragement has been the backbone in this extremely crucial, yet most exciting endeavor of my master's degree. His undying belief in me and never-say-die attitude, has been the backbone to my success to date.

In the same breath, I would like to thank Dr. Jongeun Lee, for his invaluable guidance and support in formulating the analytical model. This work would have never developed a strong foundation if it were not for his continuous and zealous interest in it.

I would also like to thank Dr. Lawrence Clark and Dr. Yu Cao for their patience and continuous support through every stage of this master's degree. I also thank my colleagues at the Compiler Microarchitecture Laboratory, Amit Pabalkar, Arun Kannan, Deepa Kannan, Rooju Chokshi, Sandeep Marathe and Sai Mylavarapu for their interesting perspectives on my research.

I am very grateful to my parents and brother Vinod, whose unflinching love, support and belief in me has made life's challenges so much more easily surmountable. I thank our Lord Jesus Christ who has been with me throughout my career, holding my hand and leading me into green pastures. To Him, I am ever grateful and dedicate this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

I. Introduction

Increasing exponentially with technology scaling, the soft error rate in earth-bound embedded systems, manufactured in deep sub-nanometer technology, is projected to become a serious design consideration. With smaller feature sizes, reduced voltage level and lower noise margins, the basic computing devices, i.e., the transistors, have become extremely prone to transient faults. A transient fault results in erroneous program states and eventually incorrect outputs, but it is temporary and non-destructive, i.e., resetting the device, restores normal behavior. While a transient fault may be caused by several sources, such as static noise in digital circuits, interconnect coupling, charge sharing noise etc., radiation particle strikes are responsible for more transient faults than all the other causes put together [44].

Considered a concern for space-bound applications, the SUN server crash in 2000 [29], opened up the possibility of serious implications of soft errors for earth-bound computing systems. Today in a high-end server, a soft error may occur every 170 hours, and a router farm will experience a networking error once every 17 hours. However, the soft error rate is projected to increase by 3 orders of magnitude as we continue to scale the feature size from present 45 nm to 22 nm in a decade. The soft error rate in desktops will increase from one per year today to about one per day. This poses a very significant threat to computing, and to the ITRS 2007 declaration of soft errors as one of the most important design concerns.

When a cosmic particle e.g., a neutron, or an alpha particle, strikes the vicinity of the gate of a digital CMOS transistor, it creates free holes and electrons in its wake [30]. If the collected charge becomes greater than some threshold, the logic state of the device can toggle. This change in the value of a transistor is called an *Upset*. If an upset causes a change in the architectural state of the processor, it is called an *Error*. Three natural masking effects prevent an upset from becoming an error, *i*) logical masking, *ii*) electrical masking, and *iii*) temporal masking. Since temporal masking is very strong in combinational logic [50], very few upsets in combinational logic circuits translate into an error. However, the absence of temporal masking in the memories, leaves the memories($L1$ cache) more sensitive to upsets. Furthermore, since memories can occupy a majority area in high-end processors, and operate at much lower voltages, they are extremely vulnerable to soft errors. In fact, according to [8, 13] more than 50% transient errors occur in memories.

Using ECC-based soft error protection mechanisms are popular and regularly deployed in lower-level memories, using any such soft error detection and prevention technique in the caches has very significant power and performance penalties [21, 27]. For example, using Single-bit Error Correction and Double-bit Error Detection (SEC-DED) codes may increase the power consumption by 22% [38], and area by 25%. While it may be possible to hide the performance penalty, it is not possible to hide the power penalty. Consequently novel techniques are required for caches that can reduce failure rates while incurring minimal power and performance overheads. Several microarchitectural techniques to prevent soft errors in caches have been proposed [9,23,33,45], but they require architectural modification for their implementation. A software solution is preferable not only because it will be applicable on any existing or future machine, but also because it allows for a dynamic trade-off of reliability for power and performance.

At the compiler-microarchitectural level, the concept of vulnerability [32] has been used to quantify the "susceptibility of cache data to soft errors". A program variable is vulnerable in cache, if it will be used by the processor, or if it will be committed to the lower levels of memory. However, if the variable will be overwritten, or it will be evicted from

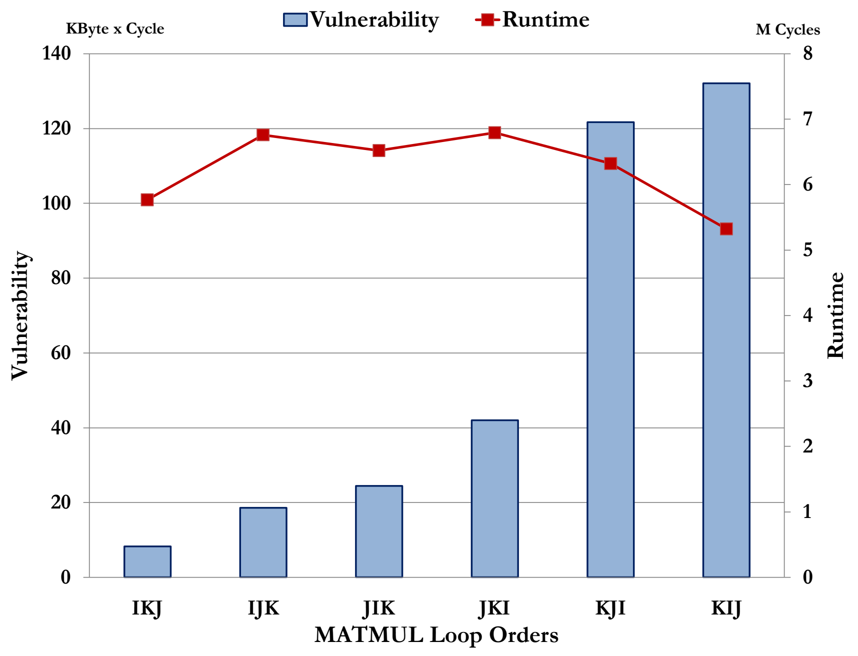


Fig. 1. Performance and vulnerability variation across different loop orders of the MATMUL program.

the cache without writing it back to the lower levels of memory (e.g., a non-dirty line in a write-back cache), then it is not vulnerable. The sum of the the vulnerable duration of all the variables is defined as program vulnerability and is a metric of failure rate due to soft errors in cache. The vulnerability of a variable is therefore dependent on when a variable is read/written/evicted. Since the compiler can change the read/write patterns of variables, it should have a significant impact on the vulnerability of a program. Figure 1 plots the runtime and vulnerability of the matrix multiplication loop for different loop orderings. The two important observations from this graph are: i) Among the possible loop-orders, the variation in vulnerability is 13X, while the variation in performance is less than 30%. This implies that significant reduction in vulnerability is possible at minimal runtime penalty. ii) The graph in Figure 1 plots the loop-orders sorted by increasing runtime, to expose the fact that runtime and vulnerability do not follow the same trends, underlying the need for developing compiler techniques for vulnerability optimization. Finally, very interesting design points, such as the IKJ loop orders exist that have low vulnerability and low runtime.

Developing compiler techniques require a method to statically estimate program vulnerability. However, only simulation-based techniques to known to compute vulnerability [45], and they are clearly not usable for a compiler technique. This work proposes a novel, accurate analytical model to estimate the vulnerability of a program, given the processor architecture specifications. This work builds upon the cache-miss equations(CME) developed by Gosh et al. [17]. Experimental validation of our methodology demonstrates about 7% inaccuracy in vulnerability estimation. Our approach is accurate enough to optimally drive important code transformations, like loop interchange, loop fusion/fission, array interleaving, and array placement.

II. Related Work

Soft errors became widely known with the introduction of DRAMs in the 1970s. In these early devices, chip packaging materials contained small amounts of radioactive contaminants, which caused soft errors. Extremely low amount of radioactive contamination is needed to keep soft errors in bay. Zeigler et al. [51] demonstrated that along with packaging material, cosmic rays are also very important cause of soft errors. In fact, neutrons are the main cause of soft errors. Since they are uncharged, they cannot disturb electron distribution on their own, but they undergo neutron capture by the nucleus of an atom in a chip, producing an unstable isotope which then, causes a soft error when it decays producing an alpha particle.

A. *Estimating Failure Rate*

Several physics-based and computer-simulation based modeling mechanisms have been proposed to predict the effect of upsets on microelectronic devices [14,15,34,46]. The simulation based techniques to estimate the failure rate of processors involves long hours of repeated simulation runs [36]. To alleviate this problem, Mukherjee et al. [6,32] introduced the concept of AVF (Architectural Vulnerability Factor) of processor elements at the microarchitectural level. The AVF of a processor element is a measure of the probability that a soft error in that architectural structure will lead to failure. The overall failure rate of the processor is then given by the product of the individual SER (Soft Error Rate) of each architectural element and its AVF number. This technique in essence is a statistical method of estimation and helps in attaining only a worst-case-estimate of the SER of the system. Sridharan et al. propose a reliability model to estimate the failure rate of the system based on the concept of *Critical Time.* Critical time is the amount of time that critical data(data being used by the processor) is present in the cache. The cumulative critical time of all the used bytes the system gives the *Vulnerability* of the system, from which we derive an accurate estimate of the overall FIT rate of the system. These techniques for estimation rely on approximations and therefore are not accurate. In this work, an analytical technique to statically estimate the impact of soft errors in a program is developed which leads to the possible development of compiler techniques to reduce the failure rate of applications.

A.1. *Circui-Level Solutions*

Buchner et al. [10] demonstrated the frequency dependence of SER in combinational logic circuits, thus indicating that with the increase in processor speed, soft error rate increases exponentially. Using the TMR(Triple Modular Redundancy) technique, which includes three functionally equivalent replicas of a logic circuit and a 2−out-of-−3 voter, [37,39] reduced soft errors to acceptable levels. However, this technique has a high hardware and power overhead, which may exceed 200%. Nicolaidis [35] proposed a technique which exploited the temporal nature of soft errors, and applied a time-grain redundancy within the clock cycle greater than the duration of transient faults, to detect soft errors. In order to remove the overheads involved in the conventional methods, [25] proposed an error masking technique, which utilized the input-output propagation delay of combinational circuits, to prevent the propagation of a transient fault through the logic circuit. However, these circuit based masking techniques include an area overhead comparable to that of the redundancy techniques.

A.2. *Microarchitecture-Level Solutions*

Error detection and error correction codes have been the primary mechanisms to detect and correct soft errors in memory systems. However, an ECC system involves high power, performance and area overheads of encoding and decoding data [21,27]. While these techniques have been popular for memories, they are not suitable for caches, since caches are very sensitive to any power and performance overheads. In order to reduce the cache susceptibility, write-through caches have been proposed. [45] propose a combined technique using selective re-fetch of cache lines from a protected $L2$ cache, and store-through cache implementations. This reduces the susceptibility of L1 cache from soft errors, but has significant impact on the IPC of the system. [23] propose the combined approach of parity and ECC codes to protect the L2/L3 caches in an area-efficient manner. The area-overhead using this technique is greatly reduced, but the performance penalty and implementation cost involved in the ECC codes cannot be avoided. [33] proposes a high-level architecture based technique to protect SRAM memories from soft errors using a combination of Reed Solomon codes and Hamming Codes. [28] proposes an adaptive error correcting scheme with early-write-back that enhances the ability to use a less powerful error protection scheme for a longer time thus reinforcing reliability with a reduction in power consumption of the system also. In [31] a lightweight error detection and correction scheme(LEDAC) has been proposed for L1 caches. This method has proved to support the necessary byte-write granularity and small latency requirements of the L1 cache with only 12% area overhead, when compared to traditional EDAC schemes which have a 62.5% area overhead. [9] propose to introduce a register file cache to efficiently reduce the failure rates. [26] proposes a novel cache architecture, called PPC (Partially Protected Cache) in which there are two caches at the same level of memory hierarchy, one protected and the other unprotected. However the challenge is in partitioning the data between the two caches in order to obtain high degree of soft error protection. [11] proposed a compiler based technique to determine the data used in the application that requires most protection and thus enable error correction techniques(ECC) for only those data elements. However, all the microarchitectural techniques propose a modification in the existing architecture/hardware, and therefore cannot be applied to existing architectures. In addition they cause an increase in design and verification complexity.

A.3. *System-Level Solutions*

Software solutions are preferred as they can be implemented on existing architectures. In addition they provide a flexible mechanism to trade-off power, performance, and vulnerability. SWIFT was proposed by Reis et al. [42] to manage redundancy by reclaiming unused instruction-level resources present during the execution of most programs to detect and correct soft errors. [19] proposed techniques on the same lines for SMT processors. In [18] a software technique to harden the application against the impact of soft errors is proposed. In this technique, additional executable assertions to check the correct execution of the program control flow is introduced. This method checks for the proper execution of the application but does not ensure data correctness and integrity of the data produced and that written into the memory.

No compiler techniques have been proposed to reduce the impact of soft errors till date, the reason being the lack of any efficient analytical technique to statically determine the impact of soft errors on a program.

III. Program and Architecture Model

A. *Program Model*

In most applications, majority of the execution time is spent within nested loops. In this context, the data access-patterns of data within these nested loops dominate any evaluation metric for that application. Therefore, we use nested loops to evaluate the vulnerability of an application. We consider only perfectly nested loops with well defined loop bounds. Our model applies only to the array access functions which are affine combinations of the enclosing loop indices, a common model for research in compiler memory analysis. Multiple references to the same array, are assumed to have access functions of the same loop indices differing by only a constant offset. We also assume that the basic block within the loop nest contains no conditional statements. In [16] the authors have evaluated the above restrictions for the SPECfp benchmarks and found that around 72% of the loops satisfied them. For our analysis, we consider each individual loop, and the total vulnerability of the program is the collective sum of the vulnerability of each array accessed within each nested loop. Our analysis is independent of the array data layout(column major or row major) and also the relative position of the data arrays in the cache(as this relative position is incorporated into the equations and thereby evaluated). Scalars can be considered as a special 1-D array with size 1.

B. *Architecture Model*

For our vulnerability model, we assume a direct mapped cache architecture on a uniprocessor model with two level memory hierarchy ($L1$ cache and main memory). In this paper, mention of the term "cache" denotes the $L1$ cache only. Writes and read operations on the data are modeled identically and the model is of fetch-on-write policy i.e., the data to be written is first brought into the cache and then written. We assume in our analysis a write back cache architecture and every dirty cache line(in which a data element is written), updates the corresponding data segment in the memory when evicted from the cache. As per the architectural model in many embedded processors, we consider an explicit eviction event, at the end of the operation of the program, on every data element present in the cache. Therefore, every dirty cache line accessed during the execution of the program is updated in the main memory at end of the program and this factor has to be included during the vulnerability calculations. Our model is also applicable to a write-through cache model with a protected write-buffer, wherein only the vulnerability due to read operations are to be considered.

The key factors that denote the size of the cache are Associativity, Block size(cache line) and the Capacity of the cache. *Block Size* or cache line(C_L) denotes the minimum size of the data structure that is transferred between the main memory and the cache. *Capacity* or the number of sets(N_S) of the cache denotes the physical number of frames of block size in which cache lines can be placed. A set in the cache can be designed to contain more than one frame and since we assume only a direct mapped cache, the associativity here is 1, meaning, a cache line can be placed in only one location in the cache and based on the size of the data accessed by the program a number of data elements will be mapped to the same location in the cache. The total size of the cache in bytes can be given as : Cache size $C_S = N_S \times 1 \times C_L$.

IV. Vulnerability Calculation

A. *Background and Terminology*

In the description of our vulnerability model we draw upon a body of research in which iteration space and reuse vectors [49] are used to analyze memory reference behavior for dependence analysis [41] or locality optimizations [49]. We build on these principles to develop a precise mechanism to estimate the vulnerability of an application.

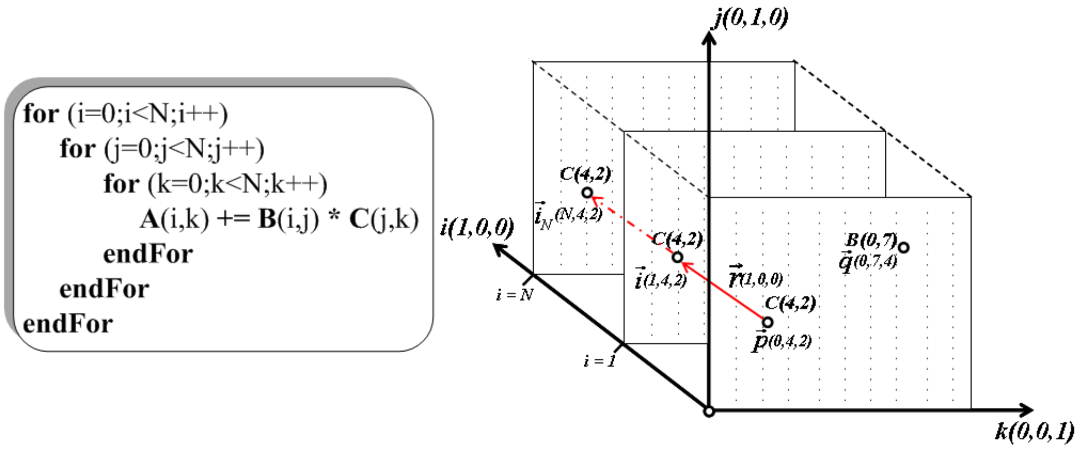
A.1. *Iteration Space*

At the compiler level, execution time can be expressed in terms of loop iterations. Therefore the execution of an n-level nested loop can be represented by a convex n-dimensional polyhedron bounded by the loop bounds [12], called *Iteration Space* in $\mathcal{Z}^n$. Each point in the iteration space represents an iteration of the innermost loop, and is called *Iteration Point.* Figure 2 shows the iteration space of the *matrix multiplication* loop nest. Every iteration point is identified by its index vector $\vec{i} = (\vec{i_1}, \vec{i_2}, ...\vec{i_n})$ where n is the depth of the enclosing nested loop structure. The outermost loop index is i_1 and i_l is the loop index of the l^{th} level loop. In Figure 2, $\vec{p} = (0, 4, 2)$ is an iteration point denoted by the iteration indices $i = 0, j = 4, k = 2$, and $\vec{i} = (1, 4, 2)$ denotes another iteration point in the iteration space. In an iteration space, if an iteration point $\vec{p_2}$ executes after $\vec{p_1}$, we say that $\vec{p_2}$ is lexicographically greater than $\vec{p_1}$. In the example in Figure 2, $\vec{q} > \vec{p}$.

A.2. *Memory Space*

A *Reference* refers to a static instance of a read or write operation in the program, while a runtime instance of this reference is called *memory access. Memory Access Function* gives the memory address that is accessed by a reference in any given iteration. For example, the memory location accessed by the reference $R = C(j, k)$ in iteration $\vec{i}$ is given by,

$$MemoryAddress_R(\vec{i}) = MemoryAccessFunction_R(\vec{i}) = BaseAddress_C + Nj + k \quad \text{(IV.1)}$$



```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        for (k=0;k<N;k++)
            A(i,k) += B(i,j) * C(j,k)
        endFor
    endFor
endFor
```

Fig. 2. Iteration space for array C in the *matmul* program. The data element $C(4,2)$ is accessed in iterations $(0,4,2),(1,4,2),...(N,4,2)$. $\vec{r}=(1,0,0)$ is the reuse vector for the reference $C(j,k)$.

A.3. *Cache Space*

In a directly mapped cache system, each memory address corresponds to a unique address in the cache. A *memory line* refers to a cache-line-sized block in the memory, whereas *cache line* refers to the actual cache-line block that a memory line is mapped to in the cache. We define the following memory functions for the reference $R = C(j,k)$:

$$MemLine_R(\vec{i}) = \lfloor MemAddr_R(\vec{i})/C_L \rfloor \tag{IV.2}$$

$$CacheAddr_R(\vec{i}) = \lfloor MemLine_R(\vec{i}) \rfloor mod N_S \tag{IV.3}$$

In Figure 2, the reference $R = C(j,k)$ accesses the data element $C(4,2)$ in iteration $\vec{i}(1,4,2)$, through the access function $MemAddr_R(\vec{i})$ where, $BaseAddress$ is the offset of the data array location in the memory relative to the other data arrays accessed. $MemLine_R(\vec{i})$ defines the memory line accessed by the iteration point $\vec{i}$, and $CacheAddr_R(\vec{i})$ defines the cache line that the particular memory line is mapped to in the cache. These memory access functions are used to determine the interference points between accesses to an array data element on a memory line in the cache.

A.4. *Reuse Vectors*

Reuse vectors express the repeated data use in a nested loop structure. If a reference accesses the same memory line in iterations $\vec{i_1}$ and then again in iteration $\vec{i_2}$, where $\vec{i_2} > \vec{i_1}$, we say that there is a reuse of data between these two iterations. In Figure 2, reference $R = C(j,k)$ accesses the data element $C(4,2)$ at iteration point $\vec{i}(1,4,2)$. From the definition of reuse vectors, the previous iteration point which accesses the same data element($C(4,2)$) can be defined by the reuse vector $\vec{r}(1,0,0)$ such that, $\vec{p} = \vec{i} - \vec{r} = (1,4,2) - (1,0,0) = (0,4,2)$.

A.5. *Cache Hit and Miss Along a Reuse Vector*

In the example loop Figure 2, three memory accesses happen in every iteration of the loop. For the reference $R = C(j,k)$, the same memory line is accessed for every iteration of the index i. Between the iterations $\vec{i_1}$ and $\vec{i_2}$, if any other reference ($A(i,k)$ or $B(i,j)$) accesses a memory line mapped to the same cache line as that of reference $R = C(j,k)$ (accessed in iteration $\vec{i_1}$), the cache lines are replaced in the cache. In this case, memory access of the reference $R = C(j,k)$ in iteration $\vec{i_2}$ is a *cache miss* (along this reuse vector $\vec{r}$), and the memory line will have to be brought in from the memory at the iteration point $\vec{i_2}$. Similarly, if an iteration $\vec{i_3}$ accesses the same cache line while it still exists in the cache (no memory transfer is required), it is said to be a *cache hit.*

B. *Read Reuse Vulnerability*

A read access resulting in a cache hit implies that the data element exists in the cache from the previous accessed iteration $\vec{p}$,($\vec{i} > \vec{p}$) till the current iteration $\vec{i}$. The variable accessed is vulnerable during this time, and this duration is a part of the *Read Reuse*

Vulnerability (RRV) of the variable. If the read access at iteration $\vec{i}$, results in a cache miss, then in a write-allocate cache, the data element is brought into the cache at iteration $\vec{i}$ itself. Thus it is not vulnerable from $\vec{p}$ to $\vec{i}$. Thus, to compute the total RRV of a variable, we need to know whether it's access incurs a cache hit or a miss.

B.1. *Identifying Cache-Hits*

A cache-miss occurs at iteration $\vec{i}$ for an array reference due to any other reference x, if the cache address accessed in iteration $\vec{i}$ matches with the cache address, of any of the array reference x, accessed by an iteration $\vec{j}$ such that $\vec{j} \in [\vec{i}, \vec{p})$. Here, $\vec{p} = \vec{i} - \vec{r}$ for the reuse vector $\vec{r}$.

Thus, the set of all iterations at which cache misses occurs for reference R, due to any other reference x is,

$$CacheMiss_R^x = \{\vec{i} | (\exists \vec{j} : CacheAddr_R(\vec{i}) = CacheAddr_x(\vec{j}) + nC_S, \vec{j} \in [\vec{p}, \vec{i}])\} \qquad \text{(IV.4)}$$

These individual sets can be combined to obtain all the iterations at which cache misses happen for a reference R, due to any other reference. Thus, $CacheMiss_R = \bigcup_x CacheMiss_R^x$.

The set of iterations which incur a cache miss for reference R is then given by the difference between the set of all the iterations constituting the iteration space(IS) and the set of cache-miss iterations($CacheMiss_R$). The vulnerability for the data elements is thus given by the product of cache-hit iterations and the size of the reuse vector $\vec{r}$.

$$RRV_R = (|IS - CacheMiss_R|) \times |\vec{r}| \qquad \text{(IV.5)}$$

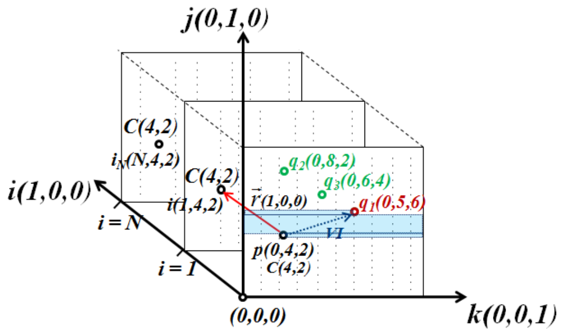


Fig. 3. The possible cache interference points ($\vec{q_1}, \vec{q_2}, \vec{q_3}$) for array element $C(4, 2)$. Cache miss at $\vec{i}$ is caused by eviction at $\vec{q_1}$ and VI denotes the vulnerable iterations due to write operation.

C. *Write Vulnerability*

We define *Write Back Vulnerability* as the time duration from the last write access of dirty line to the time when it is evicted and written back into the memory. This time is vulnerable because the dirty line will be committed to memory causing permanent change in the program state. Given a reference R with a reuse vector $\vec{r}$, we define a *cache Interference point* due to another reference x as any iteration $\vec{j}$ between $\vec{p}$, and $\vec{i} = \vec{p} + \vec{r}$, where $CacheAddr_R(\vec{i}) = CacheAddr_x(\vec{j})$.

Thus for given a reference R, the set of all cache interference points (CIP), due to any other reference x is,

$$CIP_R^x = \{i|(\exists\vec{j} : CacheAddr_R(\vec{i}) = CacheAddr_x(\vec{j}) + nC_S, \vec{j} \in [\vec{p}, \vec{i}])\} \qquad \text{(IV.6)}$$

In Figure 3, either $\vec{q_1}, \vec{q_2}$ or $\vec{q_3}$ may cause a cache miss at $\vec{i}$). Among these cache-interference points, only the first interfering iteration $\vec{q_1}$ causes replacement of the data element in the cache. The updated data element remains vulnerable only till the first cache-interference point $\vec{q_1}$(described as a shaded box in Figure 3). To compute the WBV of the datum, we need to

1. Determine the first cache interference point that causes a cache miss iteration($\vec{i}$) within the iteration space of the data array.

2. Calculate the number of iterations between the previous iteration ($\vec{p}$ or last write operation) and the identified first cache-interference-point.

C.1. *Determining First Cache-Interference Point*

To find the lexicographically first $\vec{j} \in [\vec{p}, \vec{i}), \vec{j} \in CIP$, we transform Equation (IV.6) to find out the set of iteration points $\vec{v} \in [\vec{p}, \vec{i})$, for every $\vec{i}$ such that $\vec{j} < \vec{v} < \vec{i}$. This gives all the iteration points after the first miss to the current iteration $\vec{i}$.

$$CMSubSet_R^x = \{(\vec{i}, \vec{v}) | (\exists \vec{j} : CacheAddr_R(\vec{i}) = CacheAddr_x(\vec{j}) + nC_S, \vec{j} \in [\vec{p}, \vec{i}] \ \& \ \vec{j} < \vec{v} < \vec{i})\} \tag{IV.7}$$

For each iteration $\vec{i}$, $CMSubSet_R^x$ is the set of all the iteration points tuples from the first cache miss due to any other reference x to the current iteration $\vec{i}$.

The set of all the iteration points from the first cache miss due to any reference to the current iteration can then be represented by the collection of all the individual $CMSubSet_R$ sets. Thus $CMSubSet_R = \bigcup_x CMSubSet_R^x$.

C.2. *Identifying the Vulnerable Iterations*

In addition to identifying the first cache-interference-point, Equation (IV.7) also derives a subset of iterations $\vec{v}$ that exist between the iteration causing a cache eviction and the next iteration that accesses the data element $\vec{i}$. All these iterations are not vulnerable and are to be excluded in our vulnerability calculation. In other words, from the set of iterations between the previous access $\vec{p}$ and current access $\vec{i}$, we have isolated a subset of iterations during which the data accessed in $\vec{p}$ does not exist in the cache and is not vulnerable. The set of vulnerable iterations for each cache-miss iteration $\vec{i}$ can be computed as,

$$WBV_R = |\vec{r}| - |CMSubSet_R(\vec{i}, \vec{v})| \tag{IV.8}$$

V. Reuse Vectors and Vulnerability Equations

In a cache architecture of cache-line size greater than 1 word, a memory access to one word involves memory transfer equal to the size of one cache-line. This nature of caches to fetch spatially located data elements, to reduce memory traffic and increase performance, gives rise to various interesting phenomena involving the array data access patterns. In this chapter, we introduce the plurality of reuse vectors(using the reuse analysis presented in Wolf and Lam [49]) affecting the data reuse patterns and then describe in detail their impact on our vulnerability model.

A. *Types of Reuse Vectors*

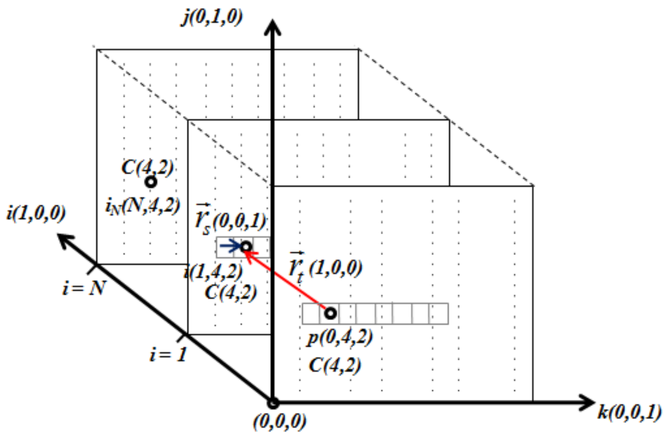


Fig. 4. Reuse vectors $\vec{r_s}$ and $\vec{r_t}$ on data element $C(4,9)$.

A.1. *Spatial Reuse Vector*

When successive iterations of a program, access successive data elements of the array(located spatially on the same memory line), the same memory line will be reused(as long as it exists in the cache), and the nature of this reuse is defined by a vector called the *spatial reuse vector*($\vec{r_s}$). In Figure 4 successive iterations of the loop index k access successive elements of the reference $C(j,k)$. The vector $\vec{r_s} = (0,0,1)$ denotes the reuse of an element $C(4,2)$ along the direction of the index k and magnitude $= 1$, because every successive iteration is an access to successive data elements on the same cache line for the array.

A.2. *Temporal Reuse Vector*

When successive accesses to the data elements of an array, reuse the memory line existing in the cache, it is called temporal reuse and is described by the *temporal reuse vector*$\vec{r_t}$. In Figure 4 the vector $\vec{r_t} = (1, 0, 0)$ denotes reuse of the element $C(4, 2)$ due to iterations $\vec{p}(0, 4, 2)$ and $\vec{i}(1, 4, 2)$. The magnitude of this vector (N^2) denotes the time in iterations between successive accesses to the same memory line.

A.3. *Group Reuse Vectors*

When the array indices of two or more references to the same array, differ by only a constant term, multiple elements of the same cache line are accessed in one iteration. Based on the magnitude of offset, more than one cache-line can also be accessed during one iteration. Since elements are accessed as groups, the data reuse pattern of the spatially located elements and the temporally accessed elements, is affected as a group. This type of reuse is called group reuse, and therefore the existing spatial and temporal reuse are transformed into their corresponding *group-spatial reuse vector* and *group-temporal reuse vector* respectively. The direction of each vector is the same as its original form, but the magnitude denotes the offset distance between the first and the last reference of the group. For example, for a group of references $D(j, k), D(j, k + 3), D(j, k + 5)$ the group-spatial reuse is $\vec{r_{gs}}(0, 0, 4)$.

A.4. *Impact on Vulnerability Equations*

In Chapter IV, the vulnerability equations had been derived assuming the existence of only one reuse vector demonstrating the data reuse pattern of the array. However, we can define a cache-miss on an iteration, only when all possible means of reuse of the cache line have failed. In other words, an iteration $\vec{i}$ can be declared to have a cache-miss, if and only if a cache-interference is realized along all possible reuse vectors for that iteration. Therefore, it is but obvious to repeat the vulnerability equations derived (Equation (IV.4)-Equation (IV.8)) for each of the reuse vectors to obtain multiple sets of cache-miss iterations and define an intersection over these sets to determine the definite set of cache-misses and cache-interference points. For a program with n_{arr} arrays and x reuse vectors affecting the data reuse, to calculate either *Read Reuse Vulnerability*(Equation (IV.4)) or *Write Back Vulnerability*(Equation (IV.7)), $(x \times n_{arr})$ equations have to be solved to obtain the resultant set of cache-interference points. This large number of equations is unacceptable for a static analysis technique and has to be optimized. For every iteration $\vec{i}$, there exists a pair $\vec{p} = \vec{i} - \vec{r}$ which defines the previous iteration that accessed the same element along the reuse vector $\vec{r}$. In order to confirm that the iteration $\vec{i}$ is a cache-miss, the presence of a cache-interference point between $\vec{i}$ and the nearest $\vec{p}$(that accessed the data last) has to be determined. The nearest $\vec{p}$ can only be defined by the smallest reuse vector and therefore using only the smallest reuse vector, the definite cache-interference points can be determined, while reducing the number of equations to be solved.

B. *Identifying the Smallest Reuse Vector*

Every data element accessed by the iteration space has its characteristic data reuse pattern and we observe that all the reuse vectors defined do not affect every element of the memory space equally. Therefore a global definition for the entire iteration space is not possible.

- An iteration($\vec{i}$) accessing the first element of a cache-line(assuming positive increments over the iteration space), is always the first access on that cache line. An iteration($\vec{p} = \vec{i} - \vec{r_s}$) accessing another element of the same cache-line cannot be defined along the vector $\vec{r_s}$ and therefore we observe that the spatial reuse vector is not valid for the set of iterations accessing the first elements of a cache-line.
- An iteration($\vec{i}$) accessing a data element for the first time during the operation, an iteration point($\vec{p} = \vec{i} - \vec{r_t}$) cannot be defined along the vector $\vec{r_t}$ and thus we deduce that the temporal reuse vector is not valid for the set of iterations accessing the elements of an array for the first time.

For each reuse vector defined, a subset of the iteration space(IS) called *domains*, can be formed in which the reuse vector is valid for memory accesses on the array for every iteration in that subset. These inherently overlapping domains can be separated into *disjoint-domains* using set theory principles: $D1 = DS \cap DT'; D2 = DS \cap DT; D3 = DS' \cap DT$; Where, DS denotes the domain in which the spatial reuse vector is valid and DT denotes that in which the temporal reuse vector is valid. On each disjoint domain, the smallest reuse vector valid over every iteration can be defined. For our analysis, we segregate the iteration space into disjoint domains based on the valid reuse vectors identified for an array, evaluate each individually and union on the resultant equations gives the set of cache-interference points for that array. It is to be noted here that, though the number of equations to be solved have increased(number of disjoint domains ¿ number of reuse vectors), by segregating the iteration space, we have reduced the exploration space for each equation and achieved adequate optimization.

C. *Derived Reuse Vectors*

Further analysis on the reuse vectors with the use of domains, indicate the presence of another reuse vector which exists between an iteration accessing the last data element of a cache-line and the iteration accessing the first element of the same memory line. This section describes the formation of this derived reuse vector and discuss its effect on the vulnerability equations.

In a program, spatial reuse defines the reuse among successive elements of a cache line and temporal reuse defines repetitive accesses to the same data element. The reuse pattern of some specific elements in the array are governed by the interference between these two reuse vectors and therefore called *derived reuse vector*. The name follows the nature of formation for this reuse vector.

For example, consider array reference $C(j,k)$ in the matmul program loop in Figure 2. Temporal reuse($\vec{r_t} = (1,0,0)$) of the reference along the index i, is realized after the spatial reuse($\vec{r_t} = (1,0,0)$) of the elements along the index k. Between access to the first element of a cache-line, on iterations i and $i+1$, all the elements of the cache line are accessed by the spatial reuse vector. An access to the last element of the cache line is the nearest previous iteration($\vec{p}$) that facilitates reuse of the cache line at the iteration $i+1$, along the temporal reuse vector. The $\vec{r_d} = (1,0,0) - C_L \times (0,0,1)$ is defined from this iteration $\vec{p}$ to the iteration($\vec{i}$) accessing the first element of the cache line. This reuse vector defining the shortest reuse vector($|\vec{r_s}| < |\vec{r_d}| < |\vec{r_t}|$) for the first elements of a cache line, is valid for the iterations accessing only the first cache-line elements and where the temporal reuse vector is valid.

$$if, \quad \vec{r_t} > \vec{r_s}, \quad \vec{r_d} = \vec{r_t} - C_L \times \vec{r_s} \tag{V.1}$$

For the array reference $B(i,j)$ in Figure 2, spatial reuse($\vec{r_s} = (0,1,0)$) of the reference along index j is realized after the temporal reuse($\vec{r_t} = (0,0,1)$) of the reference along the index k. Between successive iterations of index j, the same data element is accessed repetitively for N iterations along k, before the subsequent element on the same cache is accessed at index $j+1$. This access to the subsequent element of a cache line(by $\vec{r_s}$) is facilitated by the last temporal access to the previous element on the same cache line. The vector $\vec{r_d} = (0,1,0) - C_L \times (0,0,1)$ is defined from this nearest previous point $\vec{p}$ to the iteration($\vec{j}+1$) accessing the subsequent element on the cache line. This derived reuse vector defines the shortest reuse vector ($|\vec{r_t}| < |\vec{r_d}| < |\vec{r_s}|$) for the set of iterations for which the temporal reuse is not valid and where spatial reuse is valid.

$$if, \quad \vec{r_s} > \vec{r_t}, \quad \vec{r_d} = \vec{r_S} - C_L \times \vec{r_t} \tag{V.2}$$

D. *Time Complexity in Vulnerability Calculation*

If n_{arr} be the number of arrays in a nested loop of depth n, where m is the total number of reuse vectors identified for the array. The worst case time taken to calculate all the vulnerability equations for all the n_{arr} arrays of the program is given by $O(n_{arr}^2 \times m)$. The vulnerability equations thus generated represent a set of linear equalities or inequalities. Fast methods to solve these kind of equations for most practical loops can be found in [41]. Taking unions and intersections to find the cache-interference points takes polynomial time [22] in the number of elements of the solution sets generated. In order to determine the vulnerability of a program, integer solutions to the unions of the vulnerability equations have to be determined. The method to find the number of integer solutions in unions and intersections of closed convex polyhedrons defined by most practical scientific loops is given by [12, 47]. For the closed convex polyhedrons defined by the vulnerability equations, the method takes *polynomial time* for fixed loop bounds.

VI. Experiments

In this section, we describe in detail our experimental results validating the vulnerability equations over different cache configurations and then demonstrate the application of such a static technique through simulations, using loop kernels from the MiBench benchmark suite. For our simulations, we have modified the SimpleScalar toolset [2] to calculate the data cache vulnerability of the arrays in the program. The toolset architecture is configured to match the architecture model as described in Chapter IV. The programs are compiled using the gcc $-O3$ compiler option, in which case, the vulnerability calculated is the vulnerability of the program optimized for performance.

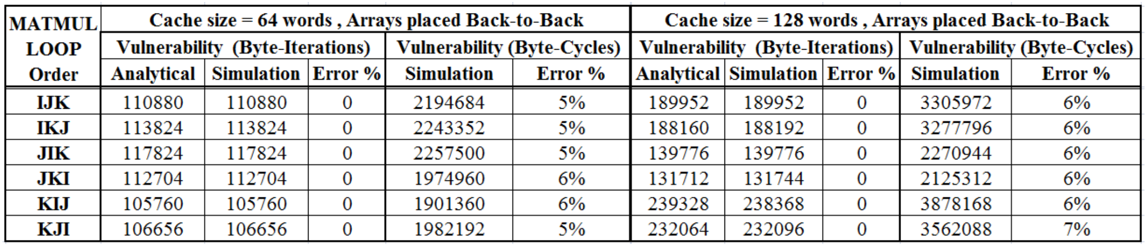
A. *Validation Experiments*

In order to demonstrate the accuracy of our analytical model, we perform experiment on the MATMUL program over different cache configurations for all the possible loop orders of the 3-dimensional nested loop. The equations derived in Chapter V evaluate the vulnerability of a program at the cache-line-level granularity(assuming every read/write operation to affect all the elements of a cache-line simultaneously) in units of loop-iterations. A modified version of the simplescalar toolset was used to simulate the vulnerability of the program in units of loop-iterations at cache-line level granularity. The analytical vulnerability values from the generated equations match these simulation values with 0% error rate(the first set of simulation values Byte-Iterations and error % in Figure 5 and Figure 6).

In order to observe a more practical applicability of our analytical model, the same program is simulated to determine the vulnerability at byte-level granularity(such that a read/write operation on any data byte in the cache-line is assumed to affect only that byte) measured in units of simulation cycles. These set of experiments on the same programs demonstrate that our analytical values differ from actual simulation values by around 5%.

It should be noted here that, for a particular cache-configuration, over all the loop orders of the matmul program, we can observe this error rate to be a constant, and therefore the use of iterations as a unit for measuring time in loop kernels is justified.

A.1. *Across Cache Sizes*

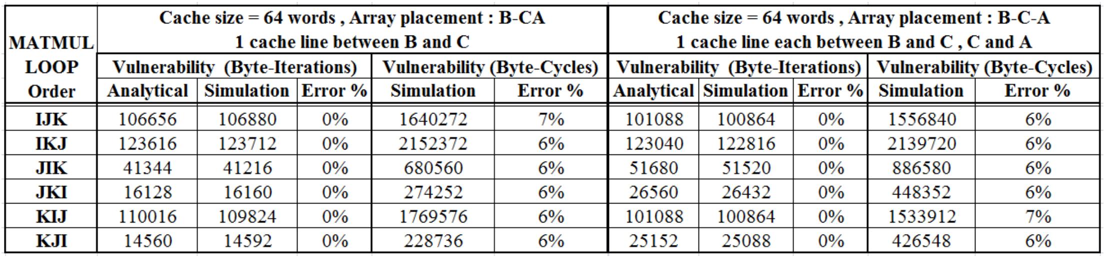


MATMUL LOOP Order	Cache size = 64 words , Arrays placed Back-to-Back					Cache size = 128 words , Arrays placed Back-to-Back				
	Vulnerability (Byte-Iterations)			Vulnerability (Byte-Cycles)		Vulnerability (Byte-Iterations)			Vulnerability (Byte-Cycles)	
	Analytical	Simulation	Error %	Simulation	Error %	Analytical	Simulation	Error %	Simulation	Error %
IJK	110880	110880	0	2194684	5%	189952	189952	0	3305972	6%
IKJ	113824	113824	0	2243352	5%	188160	188192	0	3277796	6%
JIK	117824	117824	0	2257500	5%	139776	139776	0	2270944	6%
JKI	112704	112704	0	1974960	6%	131712	131744	0	2125312	6%
KIJ	105760	105760	0	1901360	6%	239328	238368	0	3878168	6%
KJI	106656	106656	0	1982192	5%	232064	232096	0	3562088	7%

Fig. 5. Validation results of the MATMUL program across different cache sizes.

In Figure 5, analytical and simulation results of the MATMUL program over all the possible loop orders have been tabulated for two different cache sizes. The matmul program contains three 2-dimensional arrays operated over a 3-dimensional nested loop where, each loop index iterates for 8 iterations with a cache-line size of 8 words in a direct mapped cache architecture. The same program is executed over the two different cache configurations and evaluated accordingly. The cache size equal to the size of one array is chosen, so as to demonstrate cache-misses and realistic cache-replacement operations.

A.2. *Different Array Placements*



MATMUL LOOP Order	Cache size = 64 words , Array placement : B-CA 1 cache line between B and C					Cache size = 64 words , Array placement : B-C-A 1 cache line each between B and C , C and A				
	Vulnerability (Byte-Iterations)			Vulnerability (Byte-Cycles)		Vulnerability (Byte-Iterations)			Vulnerability (Byte-Cycles)	
	Analytical	Simulation	Error %	Simulation	Error %	Analytical	Simulation	Error %	Simulation	Error %
IJK	106656	106880	0%	1640272	7%	101088	100864	0%	1556840	6%
IKJ	123616	123712	0%	2152372	6%	123040	122816	0%	2139720	6%
JIK	41344	41216	0%	680560	6%	51680	51520	0%	886580	6%
JKI	16128	16160	0%	274252	6%	26560	26432	0%	448352	6%
KIJ	110016	109824	0%	1769576	6%	101088	100864	0%	1533912	7%
KJI	14560	14592	0%	228736	6%	25152	25088	0%	426548	6%

Fig. 6. Validation results of the MATMUL program for different array placement configurations.

In Figure 6, analytical and simulation results of the MATMUL program is tabulated for two different array placement configurations. In the previous experiment Figure 5, the three arrays accessed within the nested loop are physically located back to back. In this set of experiments, the physical placement of each array is altered by the inclusion of dummy arrays between the accessed arrays. In the first configuration $B - CA$, the arrays B and C are separated by a dummy array containing 8 words($= 1$ cache line). In the second configuration, another dummy array is introduced between arrays C and A. Such a configuration causes significant variation in both vulnerability and performance of the program for a direct mapped cache. In this experiment, we demonstrate the versatility of our analytical model to incorporate physical memory locations of arrays during analysis and thereby achieve accurate vulnerability estimation.

B. *Applications of the Vulnerability model*

Having established the accuracy of our analytical model for vulnerability estimation, we perform a series of experiments to demonstrate the applicability of such a static analysis technique for compiler-level optimization. With the existence of an analytical method to estimate vulnerability, at the compiler, optimal code transformations can be applied such that the program is optimized for minimal vulnerability with/or without performance tradeoff.

In this set of experiments, we perform some of the most common code transformations on loop kernels of benchmark applications and analyze the possible vulnerability variation due to these code transformations. We also analyze the performance variation due to these code transformations and determine the magnitude of possible vulnerability reduction and the corresponding performance tradeoff realized. It should be noted here, that not all transformations are applicable over all the loops, and therefore the results have been plotted for those loops wherein atleast one code transformation technique achieves significant vulnerability reduction.

B.1. *Loop Interchange Transformation*

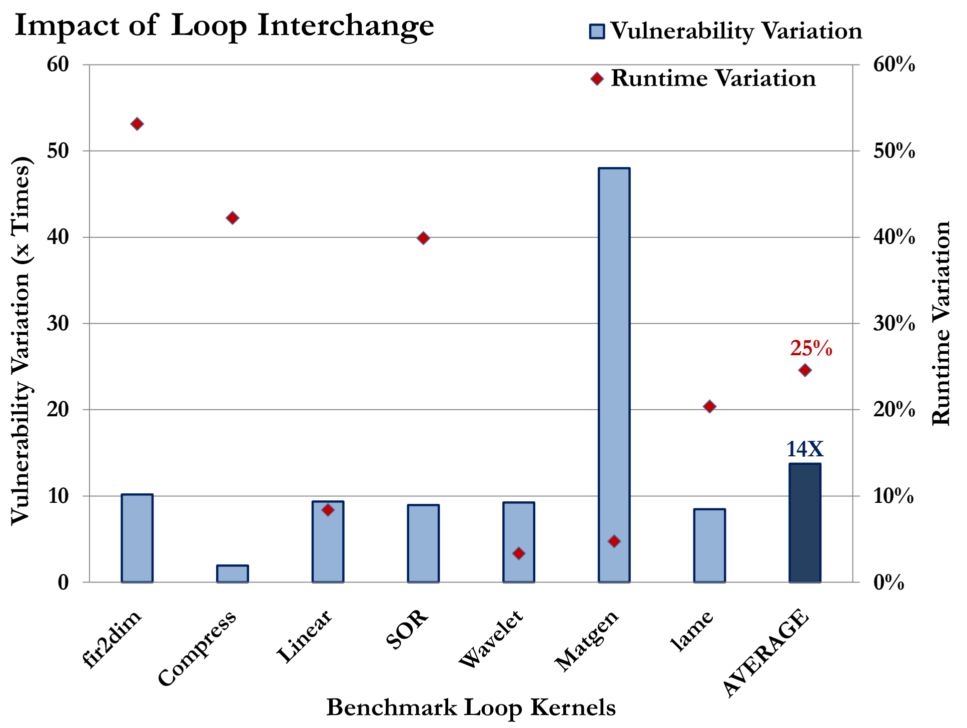


Fig. 7. Vulnerability and runtime variation for loop interchange transformation on benchmark loops.

On each loop kernel, the various possible loop interchange transformations are performed and simulated to obtain vulnerability and runtime values in each case. The graph in Figure 7 describes the vulnerability variation between loop orders with maximum and minimum vulnerability. The performance values indicate the runtime variations between these loop orders. On an average, we observe a possible $14X$ reduction in vulnerability for 25% runtime variation over the benchmark loops.

B.2. *Array Interleaving Transformation*

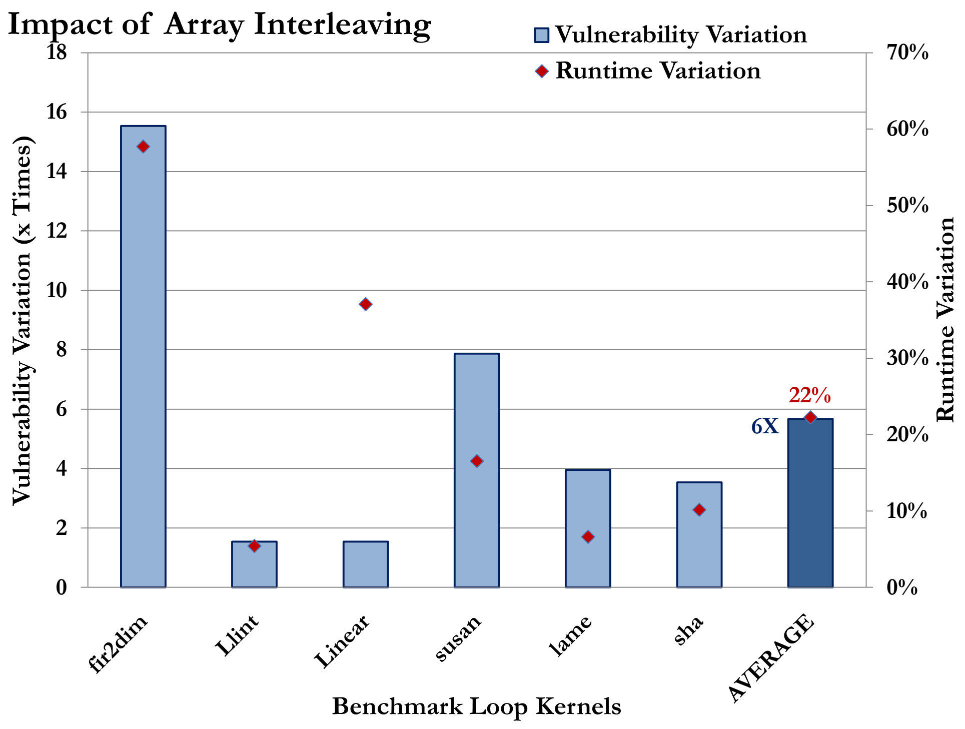


Fig. 8. Vulnerability and runtime variation for array interleaving transformation on benchmark loops.

On each loop kernel, the arrays accessed within the nested loop are analyzed for possible vulnerability variation when interleaved. The various possible cases of array interleaving are analyzed and the plot in Figure 8 describes the vulnerability and runtime variation from the base case of no interleaving on the arrays. On analysis, we observe that array interleaving is not applicable for most loops, and even when applied demonstrate only limited vulnerability reduction at heavy cost in runtime penalty.

B.3. *Loop Fission/Fusion Transformation*

On analyzing the benchmark loop kernels, and based on the nature of the data access within the loop, certain nested loops can be split to operate independently. In such a case, because only a limited number of arrays are accessed within each split loop, the array

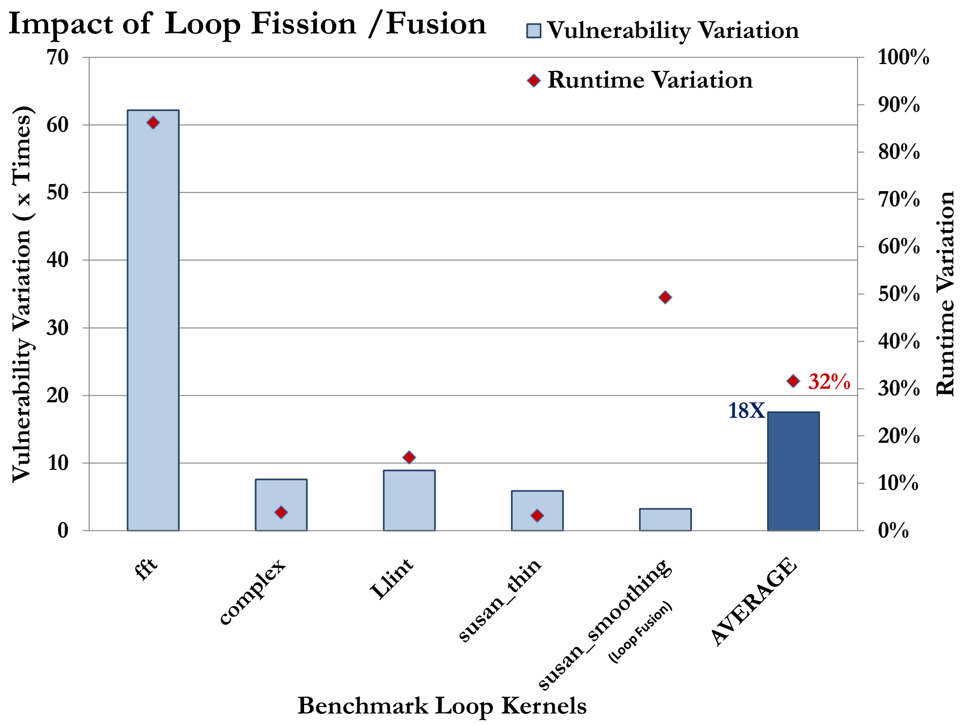


Fig. 9. Vulnerability and runtime variation for loop Fission/Fusion transformation on benchmark loops.

accesses experience less cache-interference and therefore better data reuse. In the case of *susan_smoothing* benchmark, the function with the highest runtime was analyzed and we observed the presence of two loops accessing similar data elements. The loops could be fused to access the data within a single nested loop and therefore achieve significant runtime reduction(around 50%) in addition to $3X$ vulnerability reduction. From the graph in Figure 9 we can observe that this transformation has a significant average of $18X$ vulnerability reduction which is attributed only by the high value of $62X$ for the *fft* benchmark. In general loop fission does not appear to demonstrate significant vulnerability reduction but has runtime significance. On the other hand, this transformation also has to be analyzed at the compiler so that cases similar to that of *fft* can be identified and utilized efficiently.

B.4. *Relative Array Placement*

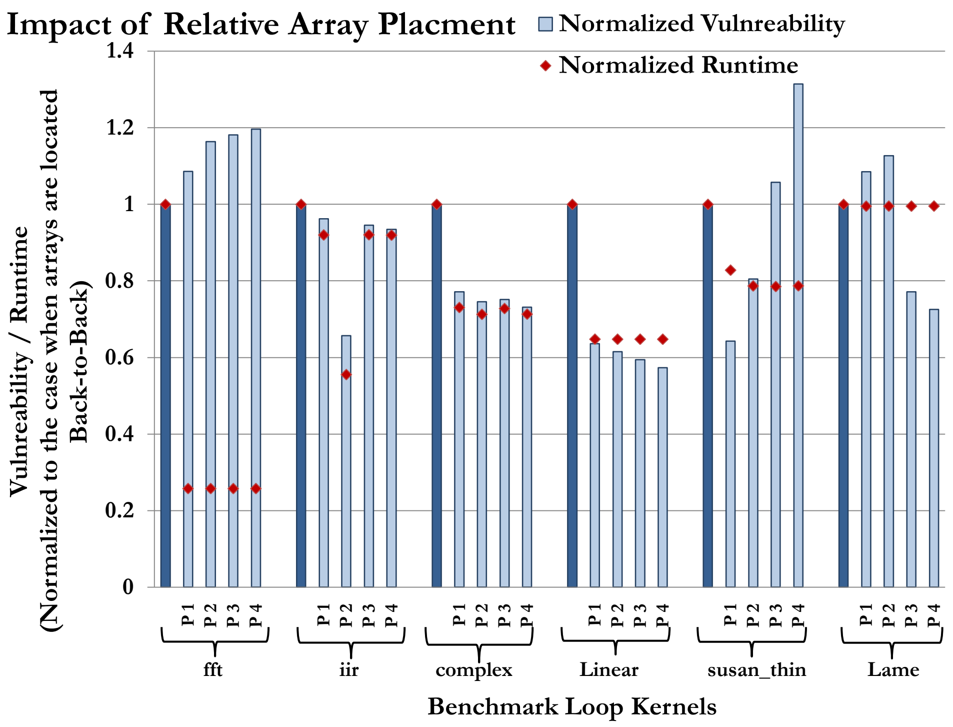


Fig. 10. Vulnerability and runtime variation in benchmark loops over different array placement configurations.

This experiment has particular significance to the underlying architecture assumed for our analysis. On a direct mapped cache, the physical location of the arrays represent the exact cache locations to which the array memory-line is mapped. On each loop kernel, we introduce dummy arrays to separate the relative positions between the arrays accessed within the loop and therefore determine the vulnerability and runtime values in each case. For each each benchmark we perform 4 relative array positions where the number of cache lines between the arrays are increased by 1 in each case. The graph in Figure 10 plots the vulnerability and runtime values of the loop normalized to the base case(where the arrays are placed back-to-back in the memory). From the analysis it can be observed that no specific trend follows the vulnerability and runtime values of the program over different array placements. This particular case, demonstrates the critical need for an analytical technique to estimate the vulnerability primarily because of time complexity involved in simulation over the entire exploration space.

VII. Conclusion

In this era of ever increasing processor complexity and ever decreasing dimensions, radiation induced soft errors threaten the reliability of embedded systems vital to mans everyday activities. At this juncture, power, area and performance tradeoffs in microarchitectural or circuit-level techniques to reduce soft errors has become unacceptable. We therefore realize the need for software level techniques to reduce the impact of soft errors in embedded applications. The critical requirement for the development of any efficient software level technique is an analytical methodology to estimate the failure rate of the program statically. In this work, an accurate static estimation technique has been developed, which evaluates the vulnerability(proportional to the failure rate) of an application. The worst-case time complexity involved in the implementation of this vulnerability model for static analysis, is comparable to existing compiler-level optimizations. We have validated the vulnerability model through simulation experiments on the MATMUL program and various benchmark loop kernels. We demonstrate the applicability of such an analytical model through experiments which indicate significant vulnerability variation achieved across various code transformation techniques with only minimal runtime variations.

REFERENCES

[1] R. Allen and K. Kennedy. "Automatic translation of FORTRAN programs to vector form". *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.

[2] T. Austin. "SimpleScalar LLC".

[3] R. Baumann. "Soft errors in advanced computer systems". *IEEE Design and Test of Computers*, pages 258–266, 2005.

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

[5] M. P. Baze, S. P. Buchner, and D. McMorrow. "A Digital CMOS Design Technique for SEU Hardening". *IEEE Trans. on Nuclear Science*, 47(6):2603–2608, Dec 2000.

[6] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. "Computing architectural vulnerability factors for address-based structures". *ISCA '05. Proceedings. 32nd International Symposium on Computer Architecture*, pages 532–543, June 2005.

[7] R. Blish. "Critical reliability challenges for the International Technology Roadmap for Semiconductors(ITRS) - technical report", Mar 2003.

[8] J. Blome, S. Mahlke, D. Bradley, and K. Flautner. "A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor". *In Proceedings of the First Workshop on Architecture Reliability*, 2005.

[9] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. "Cost-efficient soft error protection for embedded microprocessors". *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 421–431, 2006.

[10] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. "Comparison of error rates in combinational and sequential logic". *Nuclear Science, IEEE Transactions on*, 44(6):2209–2216, Dec 1997.

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

[12] P. Clauss. "Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs". *ICS '96:*

Proceedings of the 10th international conference on Supercomputing, pages 278–285, 1996.

[13] P. Dahlgren and P. Liden. "A switch-level algorithm for simulation of transients in combinational logic". *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing)*, pages 207–216, 1995.

[14] V. Degalahal, N. Vijaykrishnan, M. Irwin, S. Cetiner, F. Alim, and K. Unlu. "SESEE: A Soft Error Simulation and Estimation Engine". *Proceedings of the MAPLD International Conference*, 2004.

[15] P. Dodd and L. Massengill. "Basic mechanisms and modeling of single-event upset in digital microelectronics". *Nuclear Science, IEEE Transactions on*, 50(3):583–602, June 2003.

[16] S. Ghosh, M. Martonosi, and S. Malik. "Cache miss equations: an analytical representation of cache misses". *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 317–324, 1997.

[17] S. Ghosh, M. Martonosi, and S. Malik. "Cache miss equations: a compiler framework for analyzing and tuning memory behavior". *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[18] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. "Soft-error detection using control flow assertions". *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581–588, Nov. 2003.

[19] M. A. Gomaa and T. N. Vijaykumar. "Opportunistic Transient-Fault Detection". *Computer Architecture, International Symposium on*, 0:172–183, 2005.

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

[21] N. K. Jha. "Fault-tolerant computer system design [Book Reviews]", 1996.

[22] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. "The omega library interface guide". *Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-95-41*, 1995.

[23] S. Kim. "Area-efficient error protection for caches". *DATE 06: Proceedings of the conference on Design, automation and test in Europe*, pages 1282–1287, 2006.

[24] G. Kreisel and J. L. Krevine. *"Elements of Mathematical Logic"*. North-Holland Pub. Co., 1967.

[25] S. Krishnamohan and N. R. Mahapatra. "An efficient error-masking technique for improving the soft-error robustness of static CMOS circuits". *IEEE International SOC Conference*, pages 227–230, Sep 2004.

[26] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. "Mitigating soft error failures for multimedia applications by selective data protection". *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 411–420, 2006.

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

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

[29] D. Lyons. "Sun Scren - Sun Servers crash due to soft errors", Nov 2000.

[30] R. Mastipuram and E. C. Wee. "Soft Errors' Impact on System Reliability", Sep 2004.

[31] K. Mohr and L. Clark. Delay and area efficient first-level cache soft error detection and correction. In *ICCD*, 2006.

[32] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. "Measuring Architectural Vulnerability Factors". *IEEE Micro*, 23(6):70–75, 2003.

[33] G. Neuberger, F. de Lima, L. Carro, and R. Reis. "A multiple bit upset tolerant SRAM memory". *ACM Trans. Des. Autom. Electron. Syst.*, 8(4):577–590, 2003.

[34] H. Nguyen, Y. Yagil, N. Seifert, and M. Reitsma. "Chip-level soft error estimation method". *IEEE Transactions on Device and Materials Reliability*, 5:365–381, Sep 2005.

[35] M. Nicolaidis. "Time redundancy based soft-error tolerance to rescue nanometer technologies". *VTS'99: IEEE VLSI Test Symp.*, pages 86–94, 1999.

[36] B. Nicolescu, Y. Savaria, and R. Velazco. "Performance Evaluation and Failure Rate Prediction for the Soft Implemented Error Detection Technique". *iolts*, 00:233, 2004.

[37] A. K. Nieuwland, S. Jasarevic, and G. Jerin. "Combinational logic soft error analysis and protection". *IOLTS06*, 2006.

[38] R. Phelan. "Addressing Soft Errors in ARM Core-based Designs - technical report", 2003.

[39] D. K. Pradhan. *"Fault-tolerant computer system design"*. Prentice Hall, 1996. ISBN 0-1305-7887-8.

[40] W. Pugh. "The Omega Project".

[41] W. Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

[42] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. "SWIFT: Software Implemented Fault Tolerance". *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, 2005.

[43] P. Roche and G. Gasiot. "Impacts of front-end and middle-end process modifications on terrestrial soft error rate". *IEEE transactions on Device and Materials Reliability*, pages 382–396, Sep 2005.

[44] K. Shepard, V. Narayanan, and R. Rose. "Harmony: static noise analysis of deep submicron digital integrated circuits". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1132–1150, Aug 1999.

[45] 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.

[46] M. Sugihara, T. Ishihara, M. Muroyama, and K. Hashimoto. "A Simulation-Based Soft Error Estimation Methodology for Computer Systems". *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 196–203, 2006.

[47] S. Verdoolaege. "The Barvinok library".

[48] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. "Counting integer points in parametric polytopes using Barvinok's rational functions". *Algorithmica*, 48(1):37–66, June 2007. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41970, DOI: 10.1007/s00453-006-1231-0.

[49] M. E. Wolf and M. S. Lam. "A data locality optimizing algorithm". *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.

[50] B. Zhang, W.-S. Wang, and M. Orshansky. "FASER: Fast analysis of Soft Error Susceptibility for Cell-Based Designs". *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 755–760, 2006.

[51] J. Ziegler and W. Lanford. "The effect of sea level cosmic rays on electronic devices". *Solid-State Circuits Conference. Digest of Technical Papers. 1980 IEEE International*, XXIII:70–71, Feb 1980.