

# Shield: Cost-Effective Soft-Error Protection for Register Files

Pablo Montesinos, Wei Liu, and Josep Torrellas

University of Illinois at Urbana-Champaign

{pmontesi, liuwei, torrellas}@cs.uiuc.edu

<http://iacoma.cs.uiuc.edu>

## ABSTRACT

Device scaling and large integration increase the vulnerability of microprocessors to transient errors. One of the structures where errors can be most harmful is the register file, because it is part of the architectural state. Moreover, because the register file is frequently read, an error can quickly propagate to other parts of the processor. This paper presents *Shield*, a novel, cost-effective architecture that increases the resistance of register files to soft errors. *Shield* is based on two key observations. First, the data stored in a register is only *useful* for a small fraction of the register's lifetime. Second, not all registers are *equally* vulnerable to soft errors — a small fraction of long-lived registers is more liable to errors. *Shield* selectively protects registers by storing the ECCs of the most vulnerable registers while they contain useful data, and checks their integrity off the critical path. Our experiments show that *Shield* reduces the architectural vulnerability factor of the integer register file by up to 84% (73% on average), and by up to 100% (85% on average) for the floating-point register file while affecting performance negligibly.

## 1. Introduction

With increased chip integration levels, reduced supply voltages, and higher frequencies, transient errors are becoming a serious threat for high-performance processors. Such errors can be due to a variety of events, most notably are the impact of high-energy particles [2, 7, 23]. Since transient errors can result in serious malfunctioning [20], there have been proposals for several architectural designs that protect different structures of the processor, such as caches, memories, and datapaths [5, 13, 16, 21].

One of the critical structures to protect in a processor is the register file. It is central to the architecture of modern processors and often stores data for long periods of time. Moreover, it is read frequently, which increases the probability of spreading a faulty datum to other parts of the machine. For these reasons, some commercial processors protect their register files with either parity [3, 8] or error correcting codes (ECC) [18].

These existing approaches to protect the register file have limitations. Specifically, protecting it with only parity enables error detection but not correction. In this case, when the error is detected, recovery is only possible by

invoking a high-level operation at the OS or application level. The software might not always be able to recover from it, and it might need to terminate the application. ECC, on the other hand, enables on-the-fly detection and correction of errors. However, it does so at a cost of area, power, and possibly performance.

Given the importance of protecting the register file, our goal is to design a very cost-effective protection mechanism. We can accomplish this if we make two key observations on the use of registers in general-purpose processors. The first one is that the data stored in a physical register is only useful for a *small fraction* of the register's lifetime. The lifetime of a register is the time between register allocation and deallocation. A soft error in a register outside its useful lifetime does not have any impact on the processor's architectural state. Consequently, we only need to protect a register during its useful lifetime. The second observation is that not all the physical registers are equally vulnerable to soft errors. A small set of long-lived registers account for a large fraction of the time that registers need to be protected. The contribution of most of the other registers to the vulnerable time is very small.

Based on these two key observations, this paper proposes *Shield*, a novel architecture that provides cost-effective protection for register files against soft errors. *Shield* selectively protects the registers that contribute the most to the overall vulnerability of the register file, and only protects the *useful* lifetime of these registers. *Shield* stores the ECCs of these registers and checks their integrity off-line when they are read. We show that *Shield* requires few hardware resources. Moreover, it reduces the architectural vulnerability factor (AVF) of the integer register file by up to 84% (73% on average) and by up to 100% (85% on average) for the floating-point register file, while affecting performance negligibly.

The paper is organized as follows. Section 2 describes the motivation of this work; Sections 3 and 4 describe the design and the implementation of *Shield*; Sections 5 and 6 evaluate *Shield*; and Section 7 describes related work.

## 2. Motivation: Assigning Reliability Resources

### 2.1. Register Lifetime

Modern out-of-order processors use register renaming with a large number of physical registers to support many in-flight instructions [4]. After the processor decodes an

instruction with a destination register, it allocates a free physical register, creating a new *register version*. Later, the instruction is executed, and its result is written to the corresponding physical register. Subsequent instructions that use that value are renamed to read from that physical register. The register version is kept until the instruction that redefines the corresponding logical register retires — this is necessary to handle precise exceptions. Note that each register version is written to only *once* but can be read multiple times.

The lifetime of a register version lasts from register allocation to deallocation. Lifetime is divided into three different periods: from allocation until write; from write until last read; and from last read to deallocation. We call these periods *PreWrite*, *Useful*, and *PostLastRead*, respectively. Note that, of the whole lifetime, only the *Useful* period needs to be protected.

## 2.2. Register ACE Analysis

Errors are usually classified as undetected or detected. The former are known as Silent Data Corruption (SDC), while the latter are usually referred to as Detected Unrecoverable Errors (DUE) [10]. Errors for which detection and recovery succeeds are not treated as errors.

A structure’s *Architectural Vulnerability Factor* (AVF) is the probability that a fault in that structure will result in an error [10]. The SDC AVF and DUE AVF are the probabilities that a fault causes an SDC or a DUE error, respectively. In general, if a structure is protected by an error detection mechanism, its SDC AVF is zero. If the structure has error detection and correction capabilities, its DUE AVF is zero. In this work, we assume that the AVF for a register file is the average AVF of all its bits.

Mukherjee *et al.* [10] proposed the concept of *Architecturally Correct Execution* (ACE) to compute a structure’s AVF. ACE analysis divides a bit’s lifetime into ACE and un-ACE periods. A bit is in ACE state when a change in its value will produce an error (e.g., a wrong program output). The AVF for a single bit is the fraction of time that it is in ACE state. To calculate the total time a bit is in ACE state, we start by assuming that its whole lifetime is in ACE state, and then we remove the fraction that can be proven un-ACE. The fraction left is an upper bound on the ACE time.

As an example, Figures 1(a) and (b) show two register versions and their ACE and un-ACE periods. In both cases, a free physical register  $R$  is allocated at time  $t_a$  and deallocated at time  $t_d$ . During its *PreWrite* period, it remains in un-ACE state. At time  $t_w$ ,  $R$  is written to and, if it will be consumed at least once, it switches to ACE state. Figure 1(a) depicts a register version that is never read, so it remains in un-ACE state for its whole lifetime. On the other hand, the register version in Figure 1(b) is consumed  $n$  times, so it enters ACE state at  $t_w$  and remains in it until it is read for the last time at  $t_{rn}$ . A register version

switches back to un-ACE state during its *PostLastRead* period.

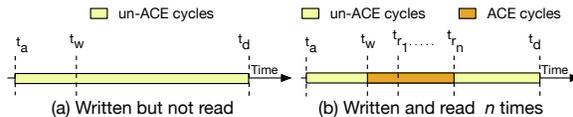


Figure 1. Examples of lifetimes of a register version.

There is one case where a register is read and it should still remain in un-ACE state. This is when the reader instructions are eventually squashed and, therefore, are never committed. For simplicity, however, in this work we do not consider this special case; if a register will be read, it is ACE.

## 2.3. Two Key Observations

Our analysis of SPECint and SPECfp 2000 applications for an out-of-order superscalar processor with 128 integer and 64 floating-point physical registers (Section 5) enables us to make two key observations.

### 2.3.1. The Combined Useful Time of All the Registers is Small

We observe that the time a register version is in *Useful* state is only a *small fraction* of the register’s lifetime. Figure 2(a) shows the average register’s *PreWrite*, *Useful*, and *PostLastRead* times for both SPECint and SPECfp applications. As shown in the figure, on average only 22% and 15% of the register lifetime is *Useful* for SPECint and SPECfp applications, respectively. Therefore, there is no need to provide protection for the whole lifetime of a register version.

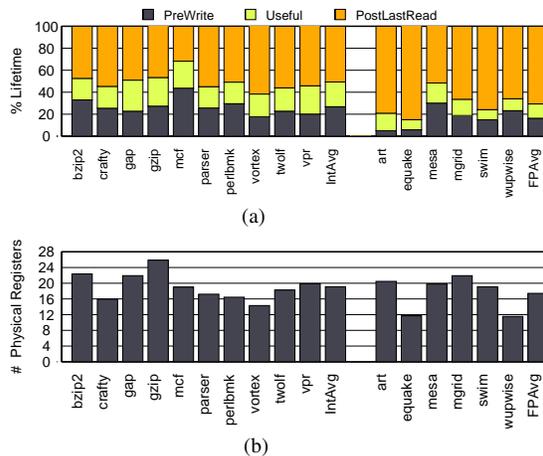


Figure 2. Lifetime breakdown for register versions (a), and average number of physical registers in useful state (b).

Figure 2(b) shows the average number of registers that are in *Useful* state at a given time. For SPECint, the average is less than 20 registers. For SPECfp, it is approximately 17.

Overall, we conclude that it is possible to reduce the vulnerability of the register file by only protecting a subset of carefully chosen registers at a time.

### 2.3.2. A Few Long-Lived Registers Provide Much of the Total Useful Time

The second observation is that not all the register versions are equally vulnerable to soft errors. A small set of long-lived registers account for a large fraction of the time that registers need to be protected. For this section only, we say that a register version is *short-lived* if by the time it is written, an instruction that reads or writes the same architectural register has been renamed. We call the other registers *long-lived*.

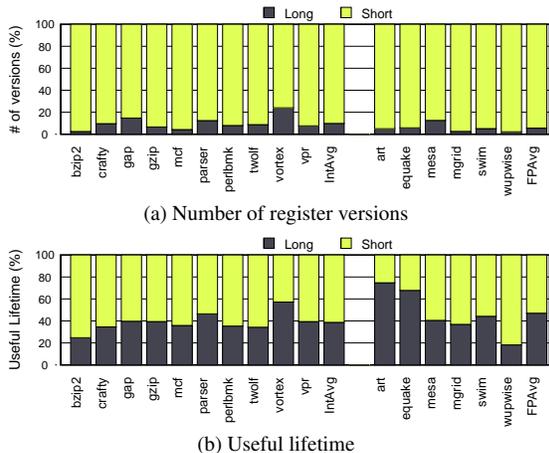


Figure 3. Characterizing short- and long-lived registers.

To see this effect, consider Figures 3(a) and 3(b). For each SPEC application, Figure 3(a) shows the percentage of long- and short-lived registers versions. On average, less than 10% of the register versions are long-lived for SPECint and SPECfp. Figure 3(b) shows the percentage of the useful lifetime that long- and short-lived versions contribute to. On average, about 40% of the contribution comes from these few long-lived register versions. Specifically, in the case of SPECfp, 5% of long-lived versions account for 46% of the useful lifetime. Therefore, it is more cost-effective to give higher protection priority to these long-lived register versions.

## 3. Shield: Protecting the Register File

### 3.1. Overview

To provide cost-effective protection for register files, we propose *Shield*. Shield performs three operations to protect the useful lifetime of a register version: (i) when the register is written, Shield generates and saves the ECC corresponding to the written data, (ii) when the register is read, Shield checks whether the register contents are still valid, and (iii) Shield keeps the protection for the register until it is read for the last time.

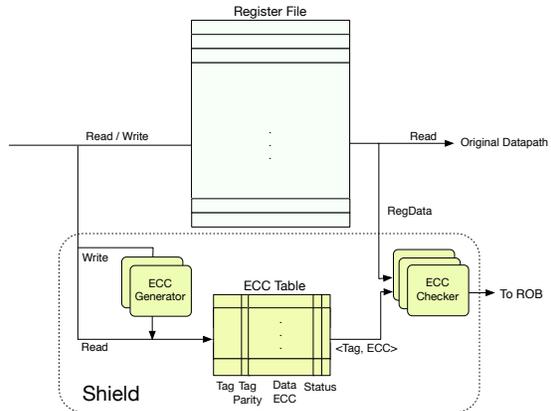


Figure 4. Architecture of Shield.

Figure 4 shows the architecture of Shield. It adds three hardware components to a traditional register file for an out-of-order microprocessor: a table that stores the ECCs of the register data, a set of ECC generators and a set of ECC checkers.

The ECC table is organized as a CAM. It protects the most vulnerable register versions in the register file. Each entry in the table protects a register version and consists of: (i) a *tag* that stores the physical register number, (ii) a *parity* bit for the tag, (iii) the *ECC* bits corresponding to the data stored in the register, and (iv) a set of *Status* bits that are used during the replacement of the ECC table entries. We will describe the replacement policy in Section 3.2.

When a physical register is about to be written, a request for protection is simultaneously sent to Shield. If Shield decides to protect the register, it tries to allocate an entry for that version. The entries in the ECC table are not pre-allocated during the register renaming stage because, as discussed in Section 2.1, there is no need to protect a register version during PreWrite time. Once an entry has been successfully allocated in the ECC table, Shield stores the physical register number in the tag. The parity bit of the tag and the ECC bits of the register data are calculated by an ECC generator in parallel with the register write operation.

Because the ECC generation takes longer than the write operation, if Shield receives a read request for an entry while its ECC is being generated, the request and ECC check operations are delayed until the ECC generation finishes.

When a physical register is read, the register file sends its data to both the datapath (e.g. ALUs) and Shield. Shield checks whether there is an entry in the ECC table whose tag matches the physical register number. If an entry is found, Shield sends the tag and its parity, and the register data and its ECC to an ECC checker verify their integrity. If an error is detected in the tag — thanks to the parity bit — the corresponding entry in the ECC table is released and the processor proceeds. If the ECC checker detects an error in the register data, the processor

stalls and takes the following actions to recover from the error: (i) it fixes the register data, (ii) flushes the ROB from the oldest instruction that reads the corrupted register version, and (iii) flushes Shield and resumes. Finally, if there is no error, the ECC checker signals no-error and the processor proceeds. The reorder buffer (ROB) is augmented with two Finish bits for the two source operand registers, respectively, to indicate that the operands have passed the check. Therefore, if no error is detected, the corresponding Finish bit is set. The Finish bit is also set when the register was not being protected by Shield by the time it was read.

When an instruction reaches the head of the ROB and is ready to retire, the Finish bits are checked. If both bits are set, the instruction can retire. Otherwise, it has to wait for the ECC checker to set the bits, or for the ROB to get full. In the latter case, the instruction is retired without taking the Finish bits into consideration in order to achieve minimal performance degradation. Our experiments show that the ROB provides enough slack for the ECC checker to verify the integrity of the data without affecting IPC.

An entry in the ECC table is deallocated and assigned to another register version when the Shield replacement algorithm decides to evict it or when a new version of the same physical register is written and sent to Shield for protection. When an entry is evicted from Shield, its associated register version will no longer be protected.

### 3.2. Entry Allocation and Replacement

When a physical register is about to be written, a request for protection for that version is sent to Shield. Shield tries to allocate an entry in the ECC table for it. If there is an entry in the table protecting a previous version of the same physical register, Shield assigns the entry to the new version. Otherwise, Shield attempts to pick a free ECC table entry. Since the ECC table is much smaller than the register file, it might not have a free entry available, and a decision has to be made either to replace an existing entry in the ECC table or to abort the allocation. Shield has to be as selective as possible at deciding which entry to replace. Replacing a recently allocated entry that protects a long-lived register to accommodate a new one that will protect a short-lived register increases the vulnerability of the system. Therefore, Shield needs to predict the lifespan of each register version.

#### 3.2.1. Predicting Short- and Long-lived Registers

A long-lived register contributes more to the register file's AVF than a short-lived one (Section 2.3.2). When Shield has to evict an used entry from the ECC table, it does not know whether the register version that it protects is still in its useful time or not.

Shield tries to evict the entry that contributes the least to the overall register file's AVF. To this end, Shield extends the short-lived register predictor proposed by Ponomarev

*et al.* [12] We first describe their approach and then how we augment it.

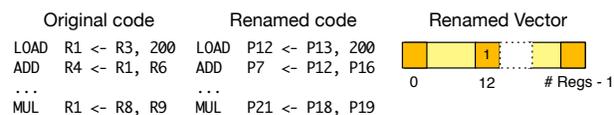


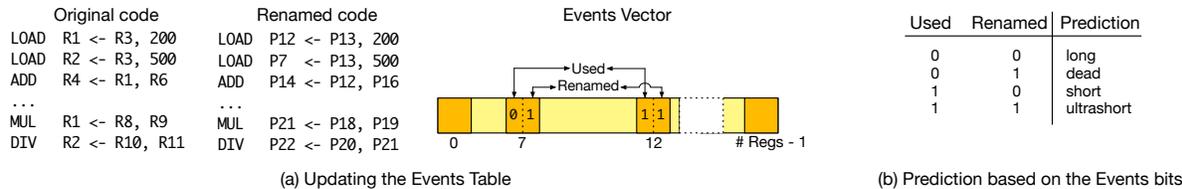
Figure 5. Predicting short-lived registers.

Figure 5 illustrates how Ponomarev *et al.*'s short-lived register predictor works. It maintains a bit vector, called *Renamed*, that has one bit per physical register. In the original code, the architectural register R1 is the destination register of the LOAD instruction. After R1 is used in the ADD instruction, the MUL instruction overwrites R1. Therefore, the MUL becomes *renamer* for the LOAD. In the renamed code, R1 has been renamed to P12. When the MUL is dispatched, it sets the bit *Renamed*[12], which is associated with physical register P12. If by the time the LOAD writes back, the *Renamed*[12] bit is set, P12 is considered to be short-lived.

Although Ponomarev *et al.*'s predictor is simple and good at predicting short-lived registers, it has a drawback. Specifically, suppose that the ADD is the only consumer of R1. In this case, we would want to consider physical register P12 as a short-lived register. However, if the LOAD has completed its write-back before the MUL goes through the rename stage, P12 will not be predicted as a short-lived register, which would be inaccurate.

In order to extend the capability of Ponomarev *et al.*'s algorithm, we reformulate the *Renamed* vector. We call it the *Events* vector, and it has two bits for each physical register, namely *Events.Renamed* and *Events.Used*. The rules for the *Events.Renamed* bit are identical to the Ponomarev *et al.*'s scheme. The new *Events.Used* bit associated with a physical register is set when renaming an instruction that consumes that physical register. Based on the *Used* and *Renamed* bits, we predict the lifespan of a register as one of the following four types: 1) *long-lived*, if both *Used* and *Renamed* bits are unset, 2) *dead*, if only the *Renamed* bit is set, 3) *short-lived*, if only the *Used* bit is set, and 4) *ultrashort-lived*, if both *Used* and *Renamed* bits are set.

Figure 6 depicts how our proposed predictor works. Similar to Ponomarev *et al.*'s algorithm, the MUL and DIV instructions act as the renamers for the first and second LOAD instruction, respectively. Therefore, the *Events.Renamed* bits of P12 and P7 are both set. Our algorithm also sets the *Events.Used* bit of P12 because R1 is used by the ADD. However, since R2 is never used, the *Events.Used* bit of P7 remains unset. The four possible combinations of the *Used* and *Renamed* bits are shown in Figure 6(b), together with the way Shield classifies the register versions. Shield separates *short* from *ultrashort* register versions. In addition, if a register version is



**Figure 6.** Predicting short- and long-lived registers.

classified as dead, its protection request is not sent to Shield.

### 3.2.2. Entry Replacement

For each protection request, Shield receives the register’s *Events* bits along with the register number and data. If there is neither an entry protecting the same physical register nor a free entry in the ECC table, Shield has to replace an existing entry or abort the allocation. Note that ECC table entries can only be allocated when a register version is written. Once a register version loses its ECC table entry, it cannot get a new one, and remains unprotected until the physical register is freed.

We propose a replacement policy that we call *Effective*. It uses the expected lifespan of a register version to select the replacement entry. It works as follows: when Shield needs to replace an entry in the ECC table, it tries to select one that protects a register version with shorter or same expected lifespan than the one to be protected. If such an entry is unavailable, Shield aborts the allocation. Table 1 shows the types of entries that can be replaced for a register version according to its prediction. For example, if a register version is predicted as short-lived, it tries to replace an entry marked as free. If a free entry is not available, it looks for an entry marked as ultrashort, then short.

We also dynamically adjust the entry status to reflect the fact that the expected lifespan gets shorter after reads. When an ultrashort or short entry is read, the status changes to free or ultrashort, respectively. The status of long entries is never changed since these entries tend to have long lifespans and may have been read many times during their lifetime.

The information about the type of register version that an ECC table entry contains is kept in the two status bits of the entry (Figure 4). Although we predict four types of register versions, we only send long-, short- and ultrashort-lived versions to the ECC table and set the status bits to *long*, *short* and *ultrashort* states, respectively. Not used entries are marked as *free*.

Prediction	Entries that can be replaced
Long	Free, Ultrashort, Short, Long
Short	Free, Ultrashort, Short
Ultrashort	Free, Ultrashort

**Table 1.** *Effective* ECC table entry replacement policy.

### 3.3. Entry Deallocation

An ECC table entry protects a given register version until the replacement algorithm reassigns the entry to another version. After a register version is read for the last time, it is effectively stale, and it is useless to protect it anymore. Ideally, Shield would like to know the time of the last read of a register version so that it can deallocate the entry as soon as it happens. However, Shield has no way of knowing whether a read is the last one. Therefore, it is possible to have stale entries in the ECC table. These stale entries hurt the efficiency of Shield since they are protecting nothing while occupying resources! The situation gets worse if these stale entries are marked as long, because they have less chance to be replaced compared to short ones.

In order to remove stale entries from the ECC table, especially the ones marked as long, we send explicit signals (called *eviction* signals) to the ECC table to indicate which entries just became stale. When the ROB sends a signal to release a physical register, this same signal is also forwarded to the ECC table as an eviction signal. The ECC table deallocates the entry whose tag matches the deallocated-register number by marking its status as free.

### 3.4. Error Recovery

Since single error correction with double error detection (SEC-DED) codes are used in this study, Shield allows the processor to recover from transient single-bit errors in the register file and detect double-bit errors. Although it may recover from some double-bit errors, in this paper we only focus on single-bit errors.

When an ECC checker detects that the register data read by instruction *I* has a single bit error, the processor stalls and enters the recovery mode. First, the checker fixes the error and writes the corrected data back to the physical register (say *P*). Second, Shield examines the ROB looking for the oldest instruction that reads *P* and flushes that instruction as well as the others that follow it. Note that only flushing the instructions that follow *I* is not enough to recover from the error. Imagine that an older instruction *J* reads *P* after *I* did, but before the error is detected. Therefore, the data that *J* read has already been corrupted. Thus, the processor has to flush from the oldest instruction that reads *P*. Finally, the ECC table is flushed and the processor can resume.

An error may happen in a mispredicted path and, although it is not necessary to recover from it because the instructions in the mispredicted path will be squashed, Shield chooses to recover anyway for simplicity. Hopefully, soft errors rarely happen and, therefore, the performance will not be affected much.

### 3.5. AVF of the Register File with Shield

Figure 7 shows different register versions and their associated ECC table entry. In Figure 7(a), Shield cannot allocate an entry for the register version and, therefore, the register remains vulnerable. If an entry was protecting a register version but is evicted before the version is read for the first time, the protection is wasted ( $t_r$  in Figure 7(b)). When the entry is evicted after the register version has been read at least once but before its last read, Shield only protects the version until the read just before the eviction (Figure 7(c)). Finally, Figure 7(d) shows a register version that has been completely protected by Shield. However, the longer an entry protects a dynamically dead register, the less efficient Shield is. By using the eviction signal described in Section 3.3, we are able to mitigate this effect.

To calculate the AVF of our system we also have to take into account the possibility of a bit flip in the ECC table or the Finish bits in the ROB. The tag in the ECC table is protected by the parity bit, and therefore a bit flip in this field can be detected. Shield then deallocates the damaged entry from the ECC table. A bit flip in the ECC field can be easily detected and corrected during the integrity check. A bit flip in the status bits will not affect the correctness of the system, only the efficiency of Shield. Thus, the AVF of the ECC table is 0.

Finally, the AVF for the Finish bits is also 0, assuming a single-bit error model. If any of the Finish bits flips to 0, that instruction will stay longer at the head of the ROB, but it will eventually retire when the ROB gets full. If any of them flips to 1, the instruction might retire before it is actually checked. However, since we are assuming that only one error can occur at a time, no other error can occur and the register data has to be correct.

## 4. Implementation Issues

### 4.1. Bypass Network

Processors use the bypass network to broadcast results from one functional unit to another so that dependent instructions can execute back to back. We have obviated the bypass network in our previous model description for simplicity. We distinguish two kinds of bypasses for a register version: (i) all its consumers read the value from the bypass network, and (ii) some of the consumers read it from the register file while others read it from the bypass network. We refer to the former as *full bypass* and to the latter as *partial bypass*.

Calculating the AVF for register versions that are fully bypassed is straightforward. Since the data stored in the register file is never used, their AVFs are zero. On the other hand, partially bypassed versions need to be protected from the time the data is written until their last non-bypassed read.

Figure 8(a) depicts the ACE and un-ACE periods of a fully bypassed register. The result of the ADD instruction is bypassed to the subsequent instructions MUL and SUB (suppose that the SUB is P1's last use) and then written into P1. Neither the MUL nor the SUB accesses the register file, hence this P1 version remains un-ACE during its entire lifetime. However, the SUB instruction in Figure 8(b) reads P1 from the register file and, therefore, P1 remains ACE until the SUB executes.

### 4.2. Accessing the ECC Table

A naive implementation of Shield requires as many read and write ports in the ECC table as the register file, and an extra write port for the eviction signals. However, with a more efficient design, it is possible to reduce the number of ports without hurting error coverage nor performance.

First, Shield is designed to check the data integrity off the critical path. Consequently it can tolerate certain latency. Even if a check from Shield is delayed for a few cycles, the performance is not likely to be affected because, usually, instructions wait in the ROB for a much longer period of time before they commit. Second, the number of ports in the register file is usually set for the worst-case scenario. Not every executed instruction reads two operands and writes one register. Moreover, in many cases, the data is bypassed and the register file is not even accessed. Therefore, we can reduce the number of read ports in the ECC table and still achieve an efficient design.

It is also possible to reduce the number of write ports required, although it is harder. There are two reasons for this: (i) registers need to be protected as soon as possible, and (ii) the eviction signals also use a write port.

We propose to send eviction signals to Shield *only* when the physical register to be freed was predicted as long-lived. The rationale is that short-lived registers are aged and evicted from the ECC table at a much faster pace than the long-lived ones. In most cases, they are evicted from the ECC table before the eviction signals are sent. Figure 3 shows that, on average, less than 10% of the register versions are considered long-lived. Therefore, technique effectively cuts down the number of signals by 90%.

In addition, we envision two ways to reduce the number of writes (i.e., protection requests) to the ECC table. First, recall that we do not send registers to the ECC table that are predicted *dead*. This avoids some protection requests. Second, we transform some protection requests to Shield into, since sending an eviction signal is much

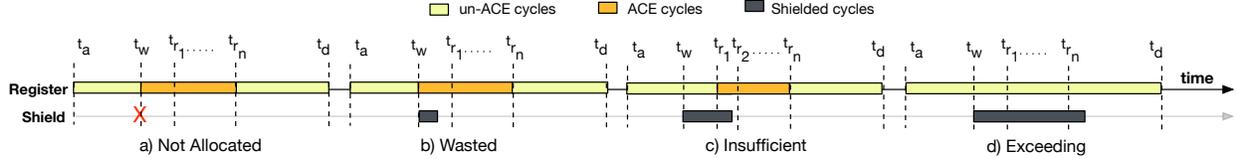


Figure 7. Computing the AVF of a register protected by Shield.

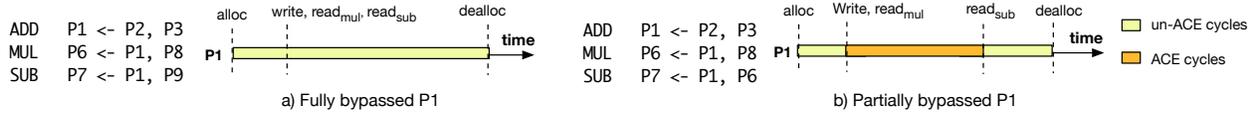


Figure 8. ACE and un-ACE periods for fully (a) and partially (b) bypassed register versions.

cheaper than sending a protection request. Specifically, when a physical register is predicted as *ultrashort* and the instructions that read it will get its value from the bypass network, an eviction signal is sent to the ECC table, instead of a protection request. Note that the eviction signal is required because there may be an ECC table entry protecting an old version of this register, which is obsolete.

### 4.3. Using More Architectural Knowledge to Improve Efficiency

Besides using the predictor described in Section 3.2.1 to guess the lifespan of registers, we leverage the usage pattern of certain architectural registers to further improve the prediction accuracy. The rationale is that some architectural registers have a specific purpose and, therefore, a specific usage pattern. For example, the *global pointer* register is written very few times during the execution of a program but is read many times and has a very long lifespan. By pinning the ECC table entry that protects the physical mapping of the global pointer until it receives the eviction signal, Shield significantly increases its efficiency.

## 5. Evaluation Methodology

We evaluate Shield using the SESC [14] cycle accurate execution-driven simulator. SESC models out-of-order processors and memory subsystems in detail. The simulated configuration is shown in Table 2. The number of ECC generators and checkers are the same as the number of write and read ports of the ECC table, respectively. 8-bit ECC codes are used for the 64-bit registers. We use an ECC table with 32 and 16 entries for integer and floating-point registers, respectively.

Shield is evaluated with the SPECint and SPECfp 2000 applications running the *Ref* data set. All of the applications are included except those that are not supported by our current framework. The applications are compiled using *gcc-3.4* with *-O3* optimization enabled. After skipping the initialization (typically 1-6 billion instructions), each application executes around 1 billion instructions.

Since applications do not run to completion, we are unable to ensure whether a register is in an ACE state or not when the simulation finishes. For example, if a simulation ends right after  $t_w$  in Figure 1(a), we would not know if the period after the write is ACE or un-ACE. To handle these edge effects, we use the *cooldown* technique that was proposed by Biswas *et al.* [1]. During the cooldown interval, we track the registers that were live at the moment that the simulation stopped. This helps us determine if a register was in an ACE or un-ACE state.

## 6. Evaluation

In this section, we first present the AVFs of Shield for both integer and floating-point register files and then assess Shield's impact on performance, area and power.

### 6.1. AVF

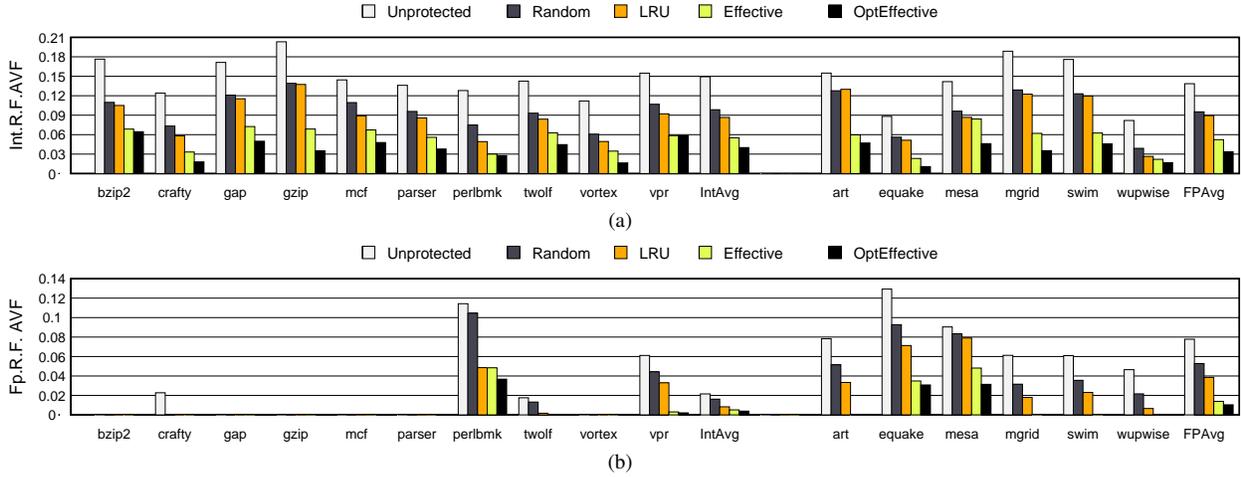
Five experiments were conducted to evaluate different replacement policies in the ECC table. The baseline policy, *Unprotected*, does not protect the register files. The other policies include the *Effective* one proposed in Section 3.2.2, the *LRU*, the *Random* and the *OptEffective* one. The latter augments the *Effective* with the *pinned* optimization described in Section 4.3.

Figures 9(a) and 9(b) show the AVFs of the integer ( $AVF_{int}$ ) and floating-point ( $AVF_{fp}$ ) register files for the different policies described. The AVFs are measured for all simulated SPECint and SPECfp applications. Since there are almost no floating-point operations in the SPECint applications, we do not discuss the  $AVF_{fp}$  for SPECint and only show it in Figure 9(b) for completeness.

Figure 9 shows that, on both integer and floating-point register files, *Effective* achieves better AVF reduction than *Random* and *LRU* for all applications. As expected, *LRU* works better than *Random* but it only reduces the  $AVF_{int}$  by 31% on average for SPECint and the  $AVF_{fp}$  by 24% for SPECfp (all relative to *Unprotected*). The reductions with *Effective* are significant. *Effective* reduces the  $AVF_{int}$  by 63% on average for SPECint and the  $AVF_{fp}$  by 42% for SPECfp relative to *Unprotected*. Finally, *OptEffective* achieves the highest level of protection, reducing the  $AVF_{int}$  up to 84% (on average 73%) for SPECint, and the

Processor		Register File		Cache & Memory		Shield	
Frequency	4 GHz	Integer:		L1 Cache:		Integer:	
Fetch/Issue/Retire	6/3/3	Size	128	Size, assoc, line	16KB, 4, 64B	ECC table size, width	32, 18 bits
ROB size	126	Width	64 bits	Latency	2 cycles	R/W ports	3/3
I-window	68	R/W ports:	6/3	L2 Cache:		ECC latency	4 cycles
LD/ST queue	48/42	FP:		Size, assoc, line	1MB, 8, 64B		
Mem/Int/FP unit	2/3/2	Size	64	Latency	12 cycles	FP:	
Branch predictor:		Width	64 bits	Memory:		ECC table size, width	16, 17 bits
Mispred. Penalty	14 cycles	R/W ports:	4/2	Latency	500 cycles	R/W ports	2/2
BTB	2K, 2-way					ECC latency	4 cycles

**Table 2.** Architecture configuration. All cycle counts are in processor cycles.



**Figure 9.** AVFs of the integer (a) and floating-point (b) register files.

$AVF_{fp}$  up to 100% (on average 85%) for SPECfp. The  $AVF_{int}$  for SPECint is 0.033 and the  $AVF_{fp}$  for SPECfp is 0.0098.

In general, Shield works better for floating-point applications because the floating-point register file has fewer registers in useful state than the integer one (Figure 2(b)), and it is easier to predict the lifespan of floating-point registers. As shown in Figure 9(b), Shield can effectively reduce the  $AVF_{fp}$  to nearly zero for *art*, *mgrid*, *swim* and *wupwise*.

## 6.2. Sensitivity Analysis

Shield increases the resistance of register files to soft errors with negligible performance degradation and modest power and area overheads. As we mentioned in Section 4.2, a naive ECC table design would require as many read ports as the register file and one write port more than the register file. However, Figure 10 shows that it is possible to use fewer ports, and still deliver similar coverage to the naive design with nearly zero performance loss. We conducted four experiments using different numbers of read and write ports in the ECC table, measuring the IPC of the programs and the AVF of the integer register file. We represent a configuration of  $x$  read ports and  $y$  write ports with the tuple  $(xR, yW)$ .

Figure 10(a) shows the impact of a reduced number of ports on the IPC of the applications. The IPCs are normalized to the *Unprotected* case, which represents a

regular register file. By performing the integrity checks off the critical path and leveraging the slack given by the ROB, Shield does not hurt performance even when the ECC table only uses half the read ports of the register file (3R, 3W). This is the design that we choose. When we further reduce the number of read ports, the performance begins to suffer. With (2R, 3W), the IPC reduces by 6.9% for SPECint and by 4.2% for SPECfp. When only one read port is used, (1R, 3W), the performance is cut by half because now the checker becomes the bottleneck and imposes a long latency that the ROB is unable to hide.

Figure 10(b) shows the impact of the reduced number of ports on the AVF of the integer register file. The AVFs are normalized to the naive design (6R, 4W). Our design choice (3R, 3W) increases the AVF by only 0.005% over the naive design. As expected, the AVF increases as we decrease the number of read ports. This data motivates our choice of the number of ECC table ports.

The choice of the ECC table size is a trade-off between area and AVF. The decision should be based on the AVF and area budget. In our design, we select 32 entries to achieve an AVF of less than 0.05 for the integer register file.

## 6.3. Power and Area Impact

Register files consume significant power in modern microprocessors. Therefore, a power-efficient protection mechanism is desired. Compared with a full ECC

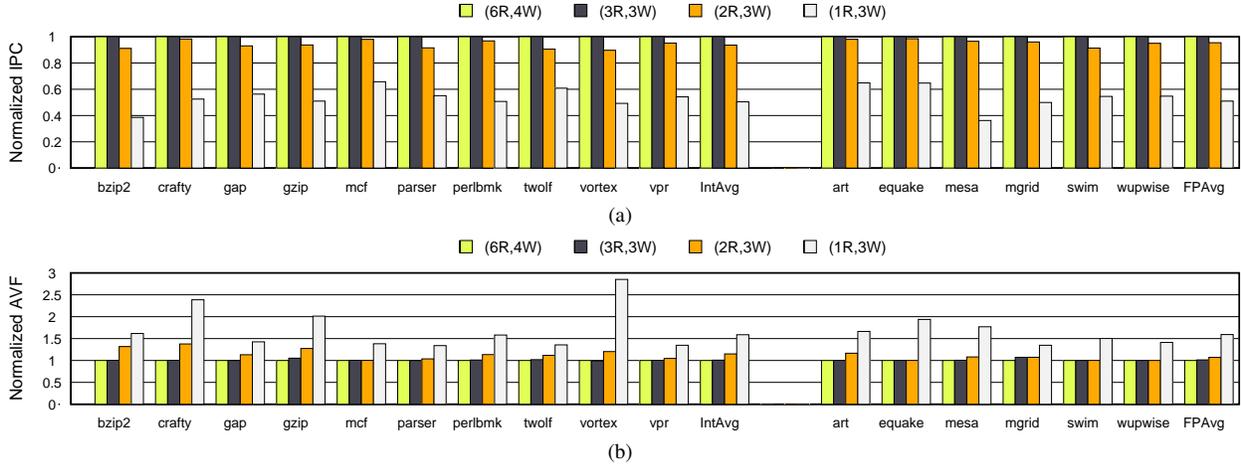


Figure 10. Impact of the number of ports in the ECC table on (a) IPC and (b) AVF of the register file

protection mechanism, Shield significantly reduces power and area costs. Unfortunately, as far as we know, there is no literature that discloses the design of a fully ECC-protected register file. Redundant pipelines and shadow register files allow the IBM G5 [18] to keep its architectural state (including the register file) in the ECC-protected *R unit*. General-purpose processors cannot afford such redundancy. Therefore, in this paper we assume a full register protection mechanism similar to the one found in the Intel Montecito [8] processor (which protects every register with parity) but augmented to use ECC. Each physical register has an instance of the ECC generation logic and a comparator (i.e. 128 ECC modules for 128 register entries). The generation logic runs continuously, so that when the register file is read, the most recently calculated ECC value can be compared against the stored ECC value.

Since Shield only uses 3 ECC generators and 3 ECC checkers, it can effectively reduce the power consumed by the ECC circuitry to less than 5% of the power of the full ECC protection. The resulting power consumption of the ECC circuitry is equivalent to about 10% of the power consumed by a plain register file according to [11]. Power is also consumed by the ECC table. It is equivalent to about 35% of the power consumed by a plain register file as we measured using a modified CACTI-3.2 [17]. Overall, Shield introduces around 45% power overhead over a plain register file. This is much less than the 2x introduced by the full ECC protection and similar to the power cost of full parity protection (47%) [11].

The computation and control logic dominates the area overhead of Shield, while the space for storing the extra check bits is secondary. For example, Montecito adds about 10% extra area overhead for the parity computation and control logic, but only about 1.5% (1/65b) and 1.2% (1/82b) extra storage for storing the parity bits for the integer and floating-point register file, respectively. The actual size of the ECC circuitry depends on the implementation, but it is roughly equal to the size of

the parity circuitry times the number of check bits [19]. If we use an 8-bit ECC to protect a 64-bit register, it approximately adds 80% area overhead for the ECC computation and control logic, and about 12.5% (8/64b) extra storage for the ECC.

Shield keeps the area overhead by using only 3 ECC generators and 3 ECC checkers. They add about  $(3 + 3)/128 \times 80\% = 3.75\%$  area overhead. The ECC table adds about 64 bytes, which is roughly 6.25% extra storage. Therefore, Shield adds about 10% area overhead, which is similar to the full parity protection mechanism described for Montecito.

## 7. Related Work

**Fully protected register files.** Traditional fault-tolerant designs protect the whole register file with parity or ECC. The IBM S/390 G5 [18] requires duplicated, lock-stepped pipelines and only instructions that produce identical results are allowed to commit and write into the ECC-protected R unit, which contains the architected state of the microprocessor. Because reading from the R unit is expensive, the G5 uses an unprotected shadow register file per pipeline. The ERC32 [3] is a SPARC processor for space applications. In addition to having parity-protected registers and buses, it also performs program flow control, which imposes an extra overhead. Like the ERC32, the Intel Montecito [8] also utilizes parity to protect the whole register file. Both the ERC32 and the Intel Montecito require software intervention to recover when a fault is detected. These designs are very expensive in terms of area, power and perhaps performance. Therefore, they are largely suitable only for mission-critical applications. In this paper, Shield selectively protects the registers that contribute the most to the overall vulnerability of the register file. Moreover, it protects them only during their *useful* lifetime to further reduce the power and overheads. In addition, Shield leverages off-line ECC verification to reduce the performance penalty.

**Partially protected register files.** In order to achieve cost-effective protection against transient errors on register files, some partial protection techniques have been proposed. Memik *et al.* [9] proposed the duplication of actively-used physical registers in unused register locations. While this approach can enhance reliability with minimal performance degradation, it can only detect errors, but not recover from them. Yan *et al.* [22] proposed a partially-protected register file. By distinguishing registers with and without ECC, the compiler's register allocation algorithm was modified to assign the most vulnerable variables to ECC-protected registers. Shield does not need to re-compile the programs because it utilizes architectural support to decrease the register file's AVF. Also, it dynamically assigns registers to an ECC table for protection.

**Register lifetime analysis.** Register lifetime has been widely studied and utilized for different optimizations. Lozano and Gao [6] used a compiler to identify short-lived variables and prevented their values from going to the register file in order to reduce the register pressure. Sangireddy and Somani [15] proposed a TriBank register file to reduce access time by exploiting the long useless states in the lifetime of a logical to physical mapping. Ponomarev *et al.* [12] proposed a small dedicated register file to cache short-lived operands in order to reduce the energy consumption in the ROB and the architectural register file. Differing with all previous proposals, Shield distinguishes ultrashort-, short- and long-lived operands, and exploits the difference to enhance the reliability of register files.

## 8. Conclusions and Future Work

Register files are very vulnerable to transient errors because they are part of the architectural state. Moreover, since registers are frequently read, errors can quickly propagate to other parts of the processor. Fully protecting register files with ECC has performance, power and area penalties. In order to solve this problem, this paper proposes *Shield*, a novel, cost-effective architecture that increases the resistance of register files to soft errors. It selectively protects registers by storing the ECCs of the most vulnerable registers while they have useful data, and checking their integrity off the critical path when registers are read. Shield reduces the AVF of the integer register file by up to 84% (73% on average) and by up to 100% (85% on average) for the floating-point register file while affecting performance negligibly. The resulting AVF for the integer register file for integer applications is 0.033, while the AVF for the floating-point register file for floating-point applications is 0.0098. Moreover, Shield only introduces a modest 10% area overhead and a 45% power overhead.

The Shield configuration presented in this paper is based on an unprotected register file. In the future, we plan to augment Shield with parity protection for the register file, as well as by adding support for multiple bit errors. We also plan to apply a similar protection mechanism to other processor structures.

## References

- [1] A. Biswas *et al.* Computing architectural vulnerability factors for address-based structures. In *ISCA*, June 2005.
- [2] E. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behavior. In *FTCS*, June 1990.
- [3] J. Gaisler. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In *FTCS*, 1997.
- [4] G. Hinton *et al.* The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 2001.
- [5] S. Kim and A. K. Somani. Area efficient architectures for information integrity in cache memories. In *ISCA*, 1999.
- [6] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *MICRO*, 1995.
- [7] W. MacKee *et al.* Cosmic ray neutron induced upsets as a major contributor to the soft error rate of current and future generation drams. *1996 IEEE Annual International Reliability Physics*, 1996.
- [8] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 2005.
- [9] G. Memik *et al.* Increasing register file immunity to transient errors. In *DATE*, 2005.
- [10] S. Mukherjee *et al.* A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, 2003.
- [11] R. Phelan. Addressing soft errors in ARM core-based designs. White Paper, 2003.
- [12] D. Ponomarev *et al.* Isolating short-lived operands for energy reduction. *IEEE Trans. Comput.*, 2004.
- [13] J. Ray *et al.* Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO*, 2001.
- [14] J. Renau *et al.* SESC simulator, January 2005. <http://sesc.sourceforge.net>.
- [15] R. Sangireddy and A. K. Somani. Exploiting quiescent states in register lifetime. In *ICCD*, 2004.
- [16] P. Shivakumar *et al.* Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, 2002.
- [17] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, 2001.
- [18] T. Slegel *et al.* IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19, 1999.
- [19] M. Tremblay and Y. Tamir. Support for fault tolerance in VLSI processors. In *International Symposium on Circuits and Systems*, 1989.
- [20] N. J. Wang *et al.* Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN*, 2004.
- [21] C. Weaver *et al.* Techniques to reduce the soft error rate of a high-performance microprocessor. In *ISCA*, 2004.
- [22] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *International Conference on Embedded Software*, 2005.
- [23] J. F. Ziegler *et al.* IBM experiments in soft fails in computer electronics (1978-1994). *IBM J. Res. Dev.*, 1996.