Using Register Lifetime Predictions to Protect Register Files Against Soft Errors*

Pablo Montesinos, Wei Liu and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
{pmontesi, liuwei, torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

To increase the resistance of register files to soft errors, this paper presents the ParShield *architecture. ParShield is based on two observations: (i) the data in a register is only useful for a small fraction of the register's lifetime, and (ii) not all registers are equally vulnerable. ParShield selectively protects registers by generating, storing, and checking the ECCs of only the most vulnerable registers while they contain useful data. In addition, it stores a parity bit for all the registers, re-using the ECC circuitry for parity generation and checking. ParShield has no SDC AVF and a small average DUE AVF of 0.040 and 0.010 for the integer and floating-point register files, respectively. ParShield consumes on average only 81% and 78% of the power of a design with full ECC for the SPECint and SPECfp applications, respectively. Finally, ParShield has no performance impact and little area requirements.*

1. Introduction

With increased chip integration levels, reduced supply voltages, and higher frequencies, soft errors are becoming a serious threat for high-performance processors. Such errors can be due to a variety of events, most notably the impact of high-energy particles [2, 8, 23]. Since soft errors can result in program visible errors [20], there have been proposals for several architectural designs that protect different structures of the processor, such as caches, memories, and datapaths [6, 13, 16, 21].

One of the critical structures to protect in a processor is the register file. It is a sizable structure that stores architectural state. It often stores data for long periods of time and 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, 9] or error correcting codes (ECC) [18]. Protecting the register file 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. Since the software might not always be able to recover from the error, the application may need to terminate. Full ECC support, on the other hand, enables on-the-fly detection and correction of errors. However, it does so at a cost in power and possibly performance.

A cost-effective protection mechanism for soft errors in register files should have no performance impact, keep the remaining Architectural Vulnerability Factor (AVF) [11] to a small value, consume modest power, and use little area. To design such mechanism, 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 not always useful. A soft error in a physical register while it is not useful will not have any impact on the processor's architectural state. Consequently, we only need to protect a register when it contains useful data. The second observation is that not all the 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 *ParShield*, a novel architecture that provides cost-effective protection for register files against soft errors. ParShield relies on the Shield concept, which selectively protects a subset of the registers by generating, storing, and checking the ECCs of only the most vulnerable registers while they contain useful data. Such support reduces the AVF of the integer register file by an average of 73% to 0.040, and the AVF of the floating-point register file by an average of 85% to 0.010. ParShield also adds a parity bit for all the registers and re-uses the ECC circuitry for parity generation and checking as well. As a result, ParShield has no Silent Data Corruption (SDC) AVF (all single-bit errors are detected), has a Detected Unrecoverable Error (DUE) AVF as low as Shield's AVF, and consumes on average only 81% and 78% of the power of a design with full ECC for the SPECint and SPECfp applications, respectively. Moreover, ParShield has no performance impact and little area requirements.

The paper is organized as follows. Section 2 describes the motivation of this work; Sections 3 and 4 describe the

*This work was supported in part by the National Science Foundation under grants CHE-0121357 and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Wei Liu is now at Intel Corporation. His email is wei.w.liu@intel.com.

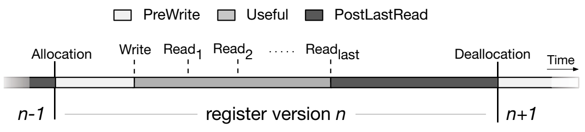


Figure 1. Lifetime of a register version.

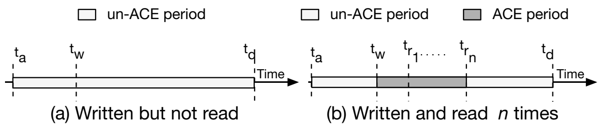


Figure 2. ACE periods of two register versions.

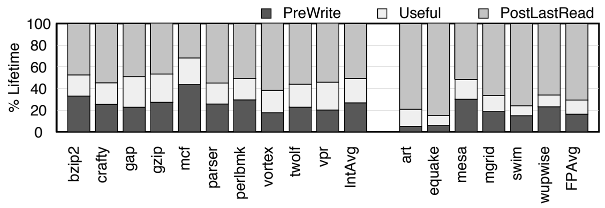


Figure 3. Integer versions lifetime breakdown.

design and the implementation of ParShield; Sections 5 and 6 evaluate ParShield; 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 a version is written to only *once* but can be read multiple times.

As Figure 1 shows, the lifetime of a register version lasts from register allocation to deallocation. We divide it 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 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) [11]. 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 [11]. 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.* [11] 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. 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 2(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 2(a) depicts a register version that is never read, so it remains in un-ACE state for its whole lifetime. Figure 2(b) shows a register version that 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 is in un-ACE state during its PostLastRead period.

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 it; 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 physical registers (Section 5) enables us to make two key observations.

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 3 shows the average integer 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 4 shows the average number of integer physical registers that are in *Useful* state at a given time. For SPECint, the average is less than 20 registers out of 128. For SPECfp, it is approximately 17 out of 128.

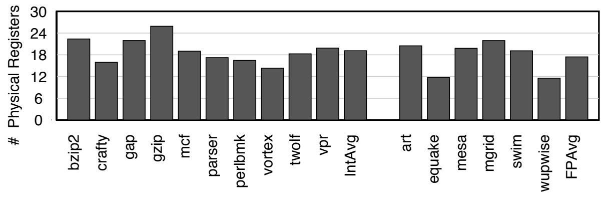
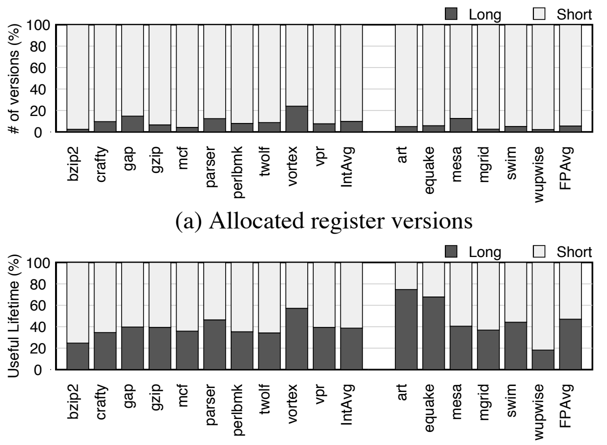


Figure 4. Average number of integer physical registers in useful state.



(a) Allocated register versions

(b) Useful lifetime

Figure 5. Short- and long-lived integer register versions characterization.

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.

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 versions 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 versions *long-lived*.

To see this effect, consider Figures 5(a) and 5(b). For each SPEC application, Figure 5(a) shows the percentage of long- and short-lived integer register versions. On average, less than 10% of the register versions are long-lived for SPECint and SPECfp. Figure 5(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 versions account for 46% of the useful lifetime. Therefore, it is cost-effective to give higher protection priority to these long-lived register versions.

3. ParShield: Protecting the Register File

To provide cost-effective protection for register files, we propose *ParShield*. ParShield is composed of (i) the *Shield* structures and (ii) the parity support. In this section, we describe the architecture, focusing mostly on Shield.

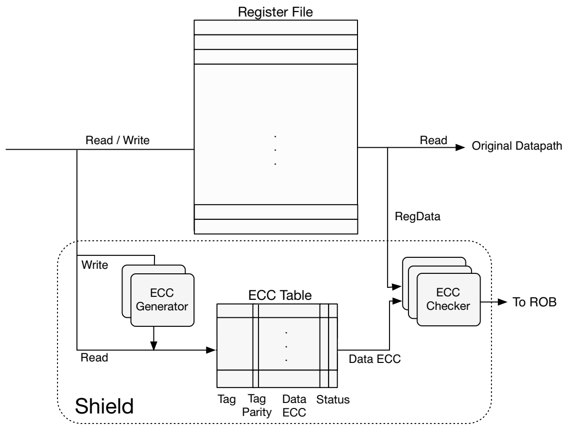


Figure 6. Shield architecture.

3.1. Shield Concept

ParShield relies on the Shield concept, which involves using ECCs to selectively protect only the subset of most vulnerable registers while they contain useful data. Shield supports three operations on one such register: (i) when the register is written, Shield generates and saves the ECC of the written data, (ii) when the register is read, Shield checks whether the register contents are still valid, and (iii) Shield keeps the ECC of the data until the register is read for the last time. Shield assumes a single-bit fault model.

Figure 6 shows the Shield architecture. It adds three hardware components to a traditional register file for an out-of-order processor: a table that stores the ECCs of some registers, 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 protects one register version and consists of: (i) a *tag* with the physical register number, (ii) a *parity* bit for the tag, (iii) the *ECC* bits of the data in the register, and (iv) a set of *Status* bits that are used during the replacement of the ECC table entries.

When a physical register is about to be written, a request for protection is 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 there is no need to protect a register version during PreWrite time. Once an entry has been successfully allocated in the ECC table, an ECC generator calculates the ECC of the register data in parallel with the register write operation.

When a physical register is read, the register file sends its data to both the datapath and Shield (Figure 6). Shield checks whether there is an entry in the ECC table whose tag matches the physical register number. If so, Shield checks the tag's parity, and sends the corresponding ECC to an ECC checker to verify the data's integrity. If an error is detected in the tag — thanks to the parity bit — the corresponding entry in the ECC table is invalidated 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) it flushes the reorder buffer (ROB) from the oldest instruction that reads the register version, and (iii) it flushes the whole ECC table and resumes. Finally, if there is no error, the ECC checker signals no error and the processor proceeds.

Each entry of the ROB is augmented with two Finish bits, one for each of the two potential source operand registers. These bits are set if, when the corresponding operand was read, it either completed the ECC check or was not protected by Shield. Single-operand instructions have one bit always set.

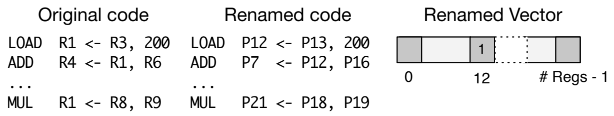
When an instruction reaches the head of the ROB and is ready to retire, the Finish bits are checked. If at least one of the bits is not set, the instruction cannot retire; it has to wait for the ECC checker to finish and set the bit(s), or for the ROB to get full. In the latter case, the instruction is retired without taking the Finish bits into consideration in order to minimize performance degradation. Our experiments show that the ROB provides enough slack for the ECC checker to verify the integrity of the data without affecting the 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 the ECC table, its associated register version will no longer be protected.

3.2. Entry Allocation and Replacement

When Shield receives a request for protection for a physical register version, it tries to allocate an entry in the ECC table. If there is an entry in the table protecting a previous version of the same physical register, Shield re-assigns the entry to the new version. Otherwise, Shield attempts to pick a free table entry. Since the table is much smaller than the register file, there may be no free entry, and a decision has to be made to either replace an existing entry in the ECC table or abort the allocation. Entry replacement has to be done carefully. 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 register versions.

3.2.1. Predicting Short- and Long-lived Registers. When Shield considers evicting an 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's goal is to evict the entry that contributes the least to the overall register file's AVF. Since a long-lived register contributes more to the register file's AVF than a short-lived one, Shield tries to evict



```
LOAD  R1 <- R3, 200   LOAD  P12 <- P13, 200
ADD   R4 <- R1, R6    ADD   P7  <- P12, P16
...                   ...
MUL   R1 <- R8, R9    MUL   P21 <- P18, P19
```

Figure 7. Predicting short-lived registers.

short-lived registers. To this end, Shield extends the short-lived register predictor proposed by Ponomarev *et al.* [12]. In the following, we first describe their approach and then how we augment it.

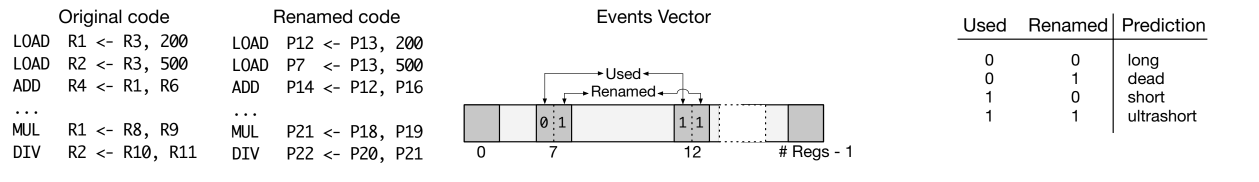
Figure 7 illustrates how Ponomarev *et al.*'s short-lived register predictor works. It maintains a bit vector, called *Renamed*, that has one bit per physical register. In Figure 7, under *original code*, a LOAD instruction loads into architectural register R1. After R1 is used in the ADD instruction, the MUL instruction overwrites R1. Therefore, the MUL is a *renamer* for the LOAD. In the *renamed code*, R1 has been renamed to P12. When the MUL is renamed, it sets the bit Renamed[12]. If by the time the LOAD loads the data into P12, the Renamed[12] bit is set, P12 is considered to be short-lived.

Although this predictor is simple and often effective, it is limited. Specifically, suppose that the ADD is the only consumer of R1. In this case, we would want to consider P12 to be a short-lived register. However, if the LOAD has loaded the data before the MUL is renamed, P12 will not be predicted as a short-lived register.

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 is set when renaming an instruction that consumes the physical register. Based on the Used and Renamed bits, when we are about to write to a physical register, we predict the register's lifespan as one of the following four types: 1) *long-lived*, if both Used and Renamed bits are reset, 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 8 shows an example of how our proposed predictor works. In the *original code*, the MUL and DIV instructions act as the renamers for the first and second LOAD instructions, 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 reset. The four possible combinations of the Used and Renamed bits are shown in Figure 8(b). *Dead* register versions are not protected by 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



```
LOAD  R1 <- R3, 200      LOAD  P12 <- P13, 200
LOAD  R2 <- R3, 500      LOAD  P7  <- P13, 500
ADD   R4 <- R1, R6       ADD   P14 <- P12, P16
...                      ...
MUL   R1 <- R8, R9       MUL   P21 <- P18, P19
DIV   R2 <- R10, R11     DIV   P22 <- P20, P21
```

Used	Renamed	Prediction
0	0	long
0	1	dead
1	0	short
1	1	ultrashort

(a) Updating the Events Vector (b) Prediction based on the Events bits

Figure 8. Predicting the lifespan of physical register versions.

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 for the rest of its lifetime.

We propose a replacement policy that we call *Effective*. It uses the expected lifespan of a register version to select the entry to replace. It works as follows: when a victim entry is needed, Shield tries to select an entry that protects a register version with a 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, it tries to replace an entry marked as free, ultrashort, and then short — in this priority order.

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

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

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

The information about the type of register version that an ECC table entry contains is kept in two Status bits (Figure 6). The four possible states of the Status bits are *long*, *short*, *ultrashort*, and *free*.

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 to a register version so that it can deallocate the entry then. 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 because they protect nothing and occupy resources. The situation is worse if these stale entries are marked as long, because they have less chance of being replaced compared to short ones.

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. If the ECC table has the corresponding entry, it marks it as free.

3.4. Error Recovery

Since we use single error correction with double error detection (SEC-DED) codes in this study, Shield allows the processor to recover from transient single-bit errors in the register file and detect double-bit errors. Although the processor may also 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 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 I and the instructions that follow it is not enough to recover from the error. Imagine that an instruction J older than I reads P after I did, but before the error is detected by the ECC checker. The data that J reads has already been corrupted. Consequently, the processor has to flush from the oldest instruction that reads P. The ECC table is then flushed so no entry in the ECC table protects one of the registers that were removed from the ROB. Finally, the processor can resume.

If an error occurs in a mispredicted path, Shield will still recover from it for simplicity.

3.5. AVF of a Register File with Shield

Figure 9 shows different physical register versions, their associated ECC table entry, and the time during which they are vulnerable to errors (ACE cycles). In Figure 9(a), Shield cannot allocate an entry for the register version and, therefore, the register is in ACE state during its whole useful lifetime. In Figure 9(b), an entry was protecting the register version but is evicted before the version is read for the first time. As a result, the register is in ACE state during its whole useful lifetime. In Figure 9(c), the entry is evicted after the register version is read at least once but before its

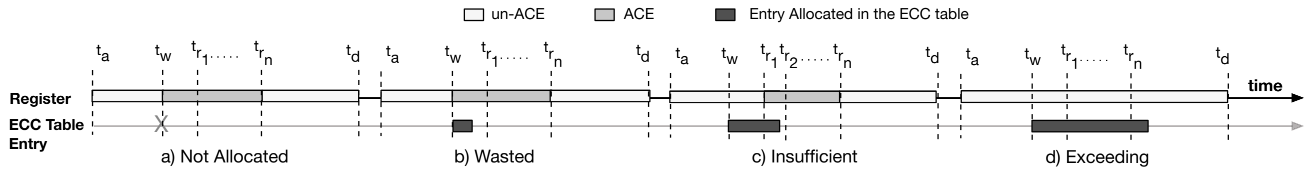


Figure 9. Computing the AVF of different physical register versions.

last read. Consequently, Shield only protects the version until the read just before the eviction. After that read, the register is in ACE state. Finally, in Figure 9(d), the ECC entry remains allocated for the whole useful lifetime of the register version. The version is completely protected and is never in ACE state. 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. Using the four cases in this figure, we can compute the ACE cycles of each register version. Since the AVF of a physical register is the fraction of ACE cycles, we can then easily compute the AVF of each physical register and of the whole register file.

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

3.6. ParShield: Shield Plus Full Register Parity

Finally, we extend Shield with storage for a parity bit for all the physical registers in the processor, and re-use the ECC circuitry for parity generation and checking as well. The result is the complete *ParShield* architecture. With the parity bit, *ParShield* reduces the SDC AVF to zero (all errors are detected) — although the DUE AVF is equal to the AVF of plain Shield (the exposure to non-correctable errors remains the same as in plain Shield). Moreover, this is accomplished at a very small cost in hardware and power.

Specifically, consider when a register write sends a protection request to the ECC table. While ParShield is checking if it should generate the data's ECC and enter them in the table — depending on the type of register version and the current contents of the ECC table — ParShield uses one ECC generator to compute the data's parity and store it in a Parity bit vector. Such operation takes a small fraction of the time taken by the generation of the full ECC.

In the same way, consider when a register read sends a request to the ECC table. While ParShield is checking if the ECC table contains the corresponding entry, ParShield reads the Parity bit vector and uses one ECC checker to check the parity. Again, this operation takes little time. Moreover, computing and checking the parity bits consumes much less power than computing and checking ECCs.

4. Implementation Issues

4.1. Bypass Network

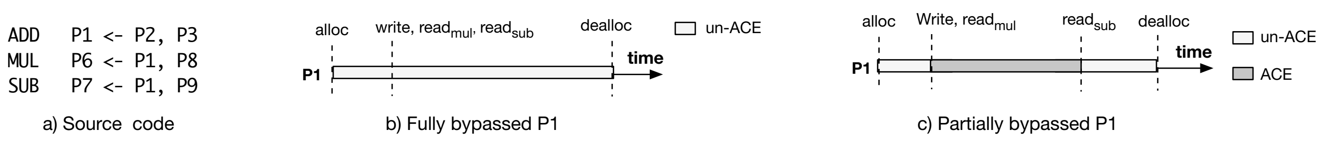
Processors use the bypass network to send results from one functional unit to another so that dependent instructions can execute back to back. We therefore need to include register bypassing in our model of AVF. We distinguish two kinds of bypasses for a register version: (i) all its consumers read the 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 10(a) shows an example where an ADD instruction generates a version of register P1, which is then read by subsequent MUL and SUB instructions. We assume that the SUB is P1's last use. Figure 10(b) shows the ACE and un-ACE periods if the result is fully bypassed. Neither the MUL nor the SUB accesses the register file. Therefore, this P1 version remains un-ACE during its entire lifetime. In Figure 10(c), the SUB instruction reads P1 from the register file and, therefore, P1 remains ACE until the SUB executes.

4.2. Accessing the ECC Table

The ECC table needs fewer ports than the register file. The reason is that the table is not as performance-critical as the register file and, therefore, does not need to be multiported for the worst case — twice as many read ports as the issue width and as many write ports as the issue width. Consequently, we reduce the number of ports and perform the ECC generation and checking off the critical path. If necessary, we support some small queueing of requests, which



```
ADD   P1 <- P2, P3
MUL   P6 <- P1, P8
SUB   P7 <- P1, P9
```

Figure 10. ACE and un-ACE periods for fully and partially bypassed register versions.

does not affect performance because instructions often wait in the ROB for a long period before committing.

In reality, there is rarely any queueing. One reason is that many instructions have fewer than two register operands. Moreover, many reads obtain their data from the bypass network and, therefore, do not access the ECC table. In addition, many register writes do not create an entry in the ECC table. Specifically, *dead* versions (Section 3.2.1), *ultrashort* versions that find the table full with non-*ultrashort* versions, and *short* versions that find the table full with *long* versions, skip ECC generation and table update.

In some cases, ParShield adds additional updates to the ECC table tags. These are caused by the eviction signals (Section 3.3). However, these signals are infrequent. They are only sent when the physical register to be freed was predicted as *long*. The rationale is that *short* and *ultrashort* register versions are aged automatically and evicted from the ECC table at a much faster pace than *long* ones — typically before the register is freed. Since Figure 5 shows that, on average, less than 10% of the register versions are *long*, the eviction signals are infrequent.

4.3. Using More Architectural Knowledge to Improve Efficiency

We extend the algorithm of Section 3.2.1 that predicts the lifespan of register versions, to leverage the usage patterns of one architectural register. The goal is to improve the prediction accuracy, given that some architectural registers have a specific purpose and, therefore, special usage patterns. Specifically, 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. Therefore, in the ECC table, we pin the entry that protects the physical mapping of the global pointer until it receives an eviction signal. Similar optimizations could be done by also considering other architectural registers.

5. Evaluation Methodology

We use a cycle-accurate execution-driven simulator [14] to model the processor and memory system architecture of Table 3. The architecture is a MIPS-like 3-issue out-of-order processor with two levels of caches, a 128-entry integer register file with 6 read and 3 write ports, and a 64-entry floating-point register file with 4 read and 2 write ports.

We evaluate the performance and the power of this architecture with the register file configurations of Table 2. *Baseline* is the architecture with no protection for the register files. *Shield* is Baseline plus the Shield architecture of Section 3.1. As shown in Table 3, the ECC table for the

Table 2. Register configurations evaluated.

Configuration	Description
Baseline	Register files with no parity or ECC protection
Shield	Baseline + Shield (Section 3.1)
ParShield	Shield + parity for all registers (Section 3.6)
FullECC	Baseline + ECC for all registers

integer register file has 32 entries and 3 read and 3 write ports; the ECC table for the floating-point register file has 16 entries and 2 read and 2 write ports. The number of ECC generators and checkers is the same as the number of write and read ports in the ECC table, respectively. *ParShield* is Shield plus the parity bit for all registers (Section 3.6). Finally, *FullECC* is Baseline plus ECC for all the 128 integer registers and 64 floating-point registers. In all cases, 8-bit ECC codes are used to protect the 64-bit registers.

We evaluate the architectures with 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 determine whether or not a register is in ACE state when the simulation finishes. For example, if a simulation ends right after t_w in Figure 2(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 ACE or un-ACE state.

6. Evaluation

In this section, we first examine the AVF results and the power and area consumption, then perform a sensitivity analysis, and finally examine register lifespan prediction.

6.1. AVF Results

We compare the AVF of Baseline to that of Shield with different replacement policies in the ECC table: *Random*, *LRU*, *Effective* (proposed in Section 3.2.2), and *OptEffective*. The latter augments *Effective* with the pinning optimization described in Section 4.3. Recall that ParShield has an SDC AVF equal to zero (all errors are detected) and a DUE AVF equal to Shield's AVF. Finally, the AVF of FullECC is zero.

Table 3. Processor and memory system modeled. Cycle counts are in processor cycles.

Processor		Register File		Cache & Memory		ParShield	
Frequency	4 GHz	Integer:		L1 Cache:		Integer:	
Fetch/Issue/Retire	6/3/3	Entries	128	Size, assoc, line	16KB, 4, 64B	ECC table entries, 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			Size, assoc, line	1MB, 8, 64B		
Mem/Int/FP unit	2/3/2	FP:		Latency	12 cycles	FP:	
Branch predictor:		Entries	64			ECC table entries, width	16, 17 bits
Mispred. Penalty	14 cycles	Width	64 bits	Memory:		R/W ports	2/2
BTB	2K, 2-way	R/W ports:	4/2	Latency	500 cycles	ECC latency	4 cycles

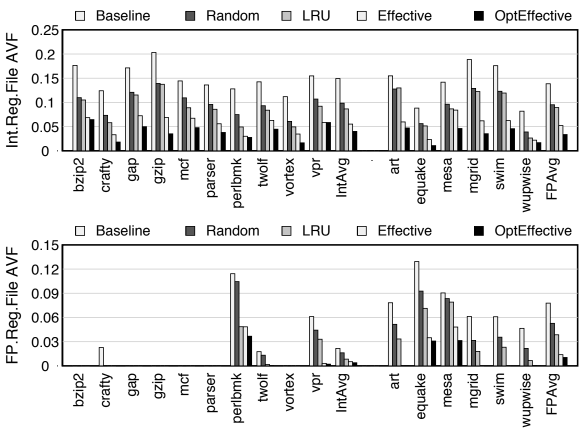


Figure 11. Integer (top) and floating-point (bottom) register file AVFs.

Figure 11 and shows the AVFs of the integer (AVF_{int}) and floating-point (AVF_{fp}) register files for the described configurations. The AVFs are shown 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 11 for completeness.

Figure 11 shows that, for all applications and on both register files, *Effective* and, especially *OptEffective*, have an AVF much lower than *Baseline*. For example, *Effective* reduces the AVF_{int} for SPECint by 63% on average and the AVF_{fp} for SPECfp by 42% on average relative to *Baseline*. *OptEffective* reduces the AVF_{int} for SPECint by up to 84% (on average 73%) and the AVF_{fp} for SPECfp by up to 100% (on average 85%) relative to *Baseline*. The resulting average AVF_{int} for SPECint is 0.040 and the average AVF_{fp} for SPECfp is 0.010. As expected, *Random* and *LRU* perform worse than chosen policies.

In general, Shield works slightly better for the floating-point register file because it has a smaller fraction of registers in useful state than the integer one. In addition, it is easier to predict the lifespan of floating-point registers. As shown in Figure 11, Shield reduces the AVF_{fp} to nearly zero for *art, mgrid, swim* and *wupwise*.

6.2. Power and Area Consumption

Register files consume a significant fraction of the power in modern processors. For example, one estimate suggests

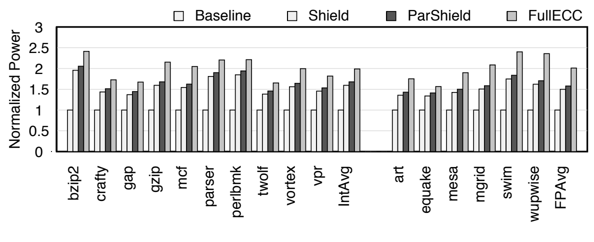


Figure 12. Integer register file power consumption.

that the integer register file consumes around 14% of the dynamic power in the processor [17]. We use CACTI 4.2 [19] to estimate the dynamic and static power of storage structures such as the register file, the ECC table, and the ECC and parity bit-fields. We use HSpice [5] models to estimate the dynamic and static power of the ECC logic.

In Figure 12, we show the total power (dynamic plus static) consumed in the integer register file for the different register configurations. For each application, the bars are normalized to *Baseline*. We do not include data for the floating-point register file because, as explained before, many of our applications do not use it much and, therefore, the average differences between configurations are small.

The figure shows that *FullECC* consumes on average 100% more power than *Baseline* for both SPECint and SPECfp applications. This is due to the combination of the ECC generators and checkers, and the additional storage for the ECC bits. With Shield, the average power is only 78% and 74% of *FullECC* for SPECint and SPECfp, respectively. This results mainly from the fewer ECC generators and checkers, and the fewer ECC operations performed — although the tags and ports in the ECC table are a significant source of power consumption.

Figure 12 also shows that *ParShield* consumes only slightly more power than Shield. The difference is small because the parity bits consume little power to generate, store and check. Overall, with *ParShield*, the average power is 81% and 78% of *FullECC* for SPECint and SPECfp, respectively. Both ParShield and Shield are more power-efficient than FullECC.

Finally, we estimate the area of the register file and the additional ECC and parity structures using CACTI 4.2. The area of the ECC logic is not added because it is negligible. Adding up the contributions of both the integer and the floating-point register files, we find that *FullECC* uses 4.9% more area than *Baseline*. Moreover, Shield and ParShield

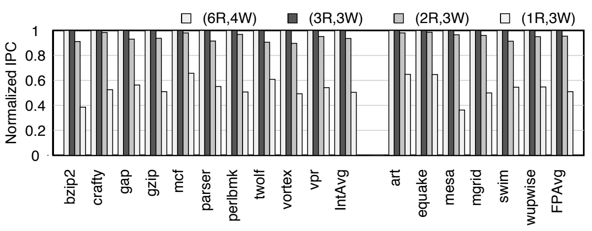


Figure 13. Impact of the number of ECC table ports on the IPC.

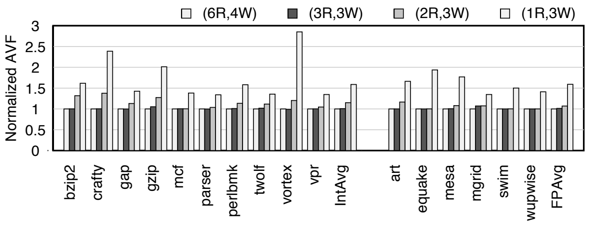


Figure 14. Impact of the number of ECC table ports on the integer register file AVF.

use 15.7% and 17.6% more area than *FullECC*, respectively. This area increase is tolerable, given the power savings provided.

6.3. Sensitivity Analysis

To gain insight into the operation of Shield, we examine the impact of the number of read and write ports in the ECC table. Figures 13 and 14 show the IPC (Instructions Per Cycle) and the integer register file AVF, respectively, when we use x read ports and y write ports (xR, yW) in the ECC table. For each application, the bars are normalized to the (6R, 4W) configuration.

Figure 13 shows that, as we go from (6R, 4W) — the same number of ports and type as in the integer register file — to (3R, 3W) — our design choice — the IPCs remain constant. This is because Shield performs the ECC generation and checks off the critical path, queues requests when necessary, and leverages the slack given by the ROB. However, when we further reduce the number of read ports, performance suffers. With (2R, 3W), the average IPC decreases by 6.9% for SPECint and by 4.2% for SPECfp. With (1R, 3W), there is a 50% performance penalty because now the ECC table becomes a major bottleneck.

Figure 14 shows the impact on the AVF of the integer register file. With our design choice (3R, 3W), the AVF changes negligibly over (6R, 4W). With (2R, 3W) and (1R, 3W), however, the AVF increases noticeably.

Overall, from this section and the previous one, we see that the (3R, 3W) design corresponds to a good tradeoff between performance, AVF, and power.

6.4. Register Lifespan Prediction

Finally, we examine the accuracy of the register lifespan predictor used in Shield (*OptEffective*). Figure 15 shows

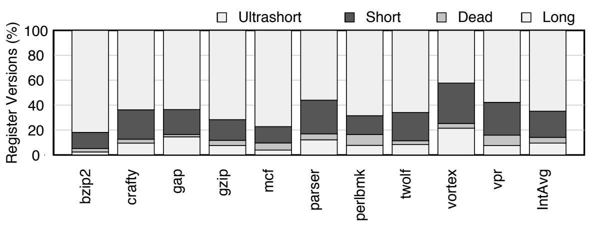


Figure 15. Breakdown of the types of register versions predicted.

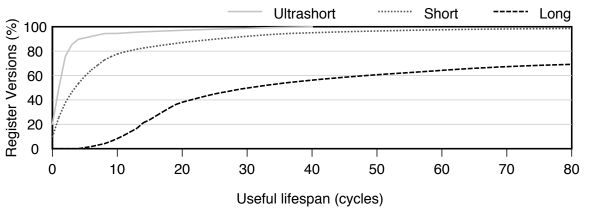


Figure 16. Cumulative distribution of useful lifespan for Ultrashort, Short, and Long registers.

the fraction of integer register versions that *OptEffective* predicts of each type. We can see that, on average, it predicts over 60% as Ultrashort and only 10% as Long. Figure 16 shows the cumulative distribution of the useful lifespan for the register versions predicted as Ultrashort, Short, and Long. From the figure, we see that *OptEffective* correctly separates the three types of registers. Indeed, most register versions predicted as Long have over 30 cycles of useful lifespan, while 95% of the register versions predicted as Ultrashort have less than 10 cycles of useful lifespan.

However, *OptEffective* does not possess oracle knowledge. Figure 17 compares Shield's AVF using *OptEffective* and an oracle algorithm (*Oracle*) for register lifespan prediction. For completeness, it also shows the AVF with Ponomarev *et al.*'s predictor (Section 3.2.1). The figure shows that, on average, *OptEffective* is very close to *Oracle*. The remaining AVF largely results from the fact that, for some parts of the applications, there are more registers in Useful state than entries in the ECC table.

7. Related Work

Fully protected register files. Traditional fault-tolerant designs protect the entire register file with parity or ECC. The IBM S/390 G5 [18] uses duplicated, lockstepped pipelines to ensure that only correct data updates the ECC-protected structure that stores the architectural state of the processor. The ERC32 [3] is a SPARC processor with parity-protected registers and buses that also performs program control flow, which imposes extra overhead. Like the ERC32, the Intel Montecito [9] 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. Our ParShield design uses parity to detect all single-bit errors, and the Shield concept to recover from most single-bit errors — by selectively protecting the

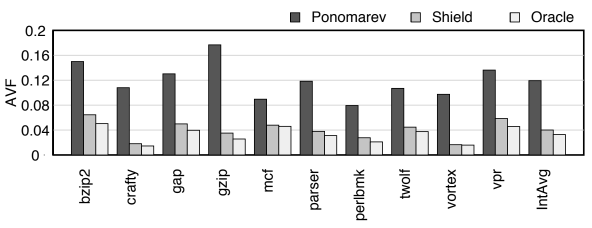


Figure 17. Comparing the AVF for different register lifespan prediction algorithms.

register versions that contribute the most to the overall vulnerability of the register file. Processor performance is not affected. The result is a design with a very cost-effective tradeoff between DUE AVF and power.

Partially protected register files. Memik *et al.* [10] proposed the duplication of actively-used physical registers in unused register locations. While their approach can enhance reliability with minimal performance degradation, it can only detect errors, but not recover from them. Yan *et al.* [22] proposed using the compiler to assign the most vulnerable variables to a set of ECC-protected registers. ParShield does not need to re-compile the programs because it offers a hardware-only solution.

Register lifetime analysis. Lozano and Gao [7] used the compiler to identify short-lived variables and prevented their values from going to the register file, thus reducing register pressure. Sangireddy and Somani [15] reduced the access time to the register file by exploiting useless periods in the register lifetime. Ponomarev *et al.* [12] used a small dedicated register file to cache short-lived operands to reduce the energy consumption in the ROB and the architectural register file. ParShield differs from all these proposals in that it proposes a hardware scheme to distinguish between ultrashort-, short- and long-lived operands, and exploits the difference to enhance the register file reliability.

8. Conclusions

Register files are vulnerable to soft errors because they are large and contain architectural state. Registers often store data for long periods of time and are read frequently, which increases the probability of spreading a faulty datum. A cost-effective protection mechanism for soft errors in register files should have no performance impact, keep the remaining AVF to a small value, consume modest power, and use little area.

In this paper, we have proposed one such mechanism, namely the *ParShield* design. ParShield relies on the Shield concept, which selectively protects a subset of the registers by generating, storing, and checking the ECCs of only the most vulnerable registers while they have useful data. Shield reduces the AVF of the integer register file by an average of 73% to 0.040, and the AVF of the floating-point register file by an average of 85% to 0.010. ParShield extends Shield with a parity bit for all the registers and the re-use of the ECC circuitry for parity generation and checking as well. As a result, ParShield has no SDC AVF (all single-bit errors are detected), has a DUE AVF as low as Shield's AVF, and consumes on average only 81% and 78% of the power of a design with full ECC for the SPECint and SPECfp applications, respectively. Moreover, ParShield has no performance impact and little area requirements.

References

[1] A. Biswas et al. Computing architectural vulnerability factors for address-based structures. In *International Symposium on Computer Architecture*, June 2005.
[2] E. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behavior. In *International Symposium on Fault-Tolerant Computing*, June 1990.
[3] J. Gaisler. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In *International Symposium on Fault-Tolerant Computing*, 1997.
[4] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 2001.
[5] Hspice User's Manual. Applications and examples, 1996.
[6] S. Kim and A. K. Somani. Area efficient architectures for information integrity in cache memories. In *International Symposium on Computer Architecture*, 1999.
[7] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *International Symposium on Microarchitecture*, 1995.
[8] 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.
[9] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 2005.
[10] G. Memik et al. Increasing register file immunity to transient errors. In *Design, Automation and Test in Europe*, 2005.
[11] S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *International Symposium on Microarchitecture*, 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 *International Symposium on Microarchitecture*, 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 *International Conference on Dependable Systems and Networks*, 2002.
[17] K. Skadron et al. Temperature-aware microarchitecture. In *International Symposium on Computer Architecture*, 2003.
[18] T. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19, 1999.
[19] D. Tarjan et al. CACTI 4.0. *Tech Report HPL-2006-86*, 2006.
[20] N. J. Wang et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks*, 2004.
[21] C. Weaver et al. Techniques to reduce the soft error rate of a high-performance microprocessor. In *International Symposium on Computer Architecture*, 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.