

# SoftECC : A System for Software Memory Integrity Checking

by

Dave Dopson

Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Electrical Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Sept 2005

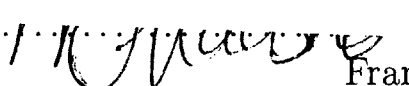
© Dave Dopson, MMV. All rights reserved.

The author hereby grants to MIT permission to reproduce and  
distribute publicly paper and electronic copies of this thesis document  
in whole or in part.

Author ..... 

Department of Electrical Engineering and Computer Science

Sept 3, 2005

Certified by ..... 

Frans Kaashoek

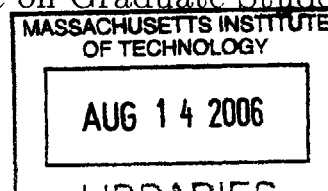
Professor

Thesis Supervisor

Accepted by ..... 

Arthur C. Smith

Chairman, Department Committee on Graduate Students



ARCHIVES

# SoftECC : A System for Software Memory Integrity Checking

by

Dave Dopson

Submitted to the Department of Electrical Engineering and Computer Science  
on Sept 3, 2005, in partial fulfillment of the  
requirements for the degrees of  
Bachelor of Science in Computer Science and Electrical Engineering  
and  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

**SoftECC** is software memory integrity checking agent. **SoftECC** repeatedly computes page-level checksums as an efficient means to verify that a page's contents have not changed. Memory errors that occur between two checksum computations will cause the two checksum values to disagree. Legitimate memory writes also cause a change in checksum value, so a page can only be protected during periods of time when it is not being written to. Preliminary measurements with an implementation of **SoftECC** in the JOS kernel on the x86 architecture show that **SoftECC** can halve the number of undetectable soft errors using minimal compute time.

Thesis Supervisor: Frans Kaashoek  
Title: Professor

## Acknowledgments

I would like to thank Frans Kaashoek for overseeing this project and providing extensive feedback on my writing. I would also like to thank him for allowing me to undertake this research in the first place and ensuring I always had the resources necessary to complete my work.

I also owe a great debt of gratitude to Chris Lesniewski-Laas for conceiving the original project idea and providing me with many hours of fruitful discussion.

The completion of this project happened to coincide quite poorly with the date for my move from Boston to Seattle. Without Mayra's invaluable assistance in packing, I would have had to choose between my belongings and my thesis.

Finally, I would like to thank my parents. Without their continual support and encouragement, I would never have made it this far.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Approach . . . . .	10
1.2	Vulnerability . . . . .	11
1.3	Contribution . . . . .	12
1.4	Rest of Thesis . . . . .	13
<b>2</b>	<b>Background and Related Work</b>	<b>14</b>
2.1	Types of Memory Errors . . . . .	14
2.2	ECC memory: the hardware solution . . . . .	15
2.3	Software Solutions . . . . .	16
<b>3</b>	<b>Design</b>	<b>18</b>
3.1	State Transitions . . . . .	19
3.2	The <b>Trapwrite</b> State . . . . .	20
3.3	The <b>Trapall</b> State . . . . .	21
3.4	Design Alternatives . . . . .	21
3.4.1	The <b>Checksum</b> Queue . . . . .	21

3.4.2	Linux: Swapfile of a modified ramfs partition . . . . .	22
<b>4</b>	<b>Implementation</b>	<b>23</b>
4.1	Implementing Page State Transitions . . . . .	23
4.1.1	Read and Write Trap Handlers . . . . .	25
4.2	The JOSkern VMM extensions . . . . .	27
4.2.1	Trapping Memory Access on x86 . . . . .	28
4.2.2	Checking for Prior Memory Access on x86 . . . . .	30
4.3	Checksums . . . . .	32
4.4	Redundancy for Error Correction . . . . .	34
4.4.1	Full Memory Copy . . . . .	36
4.4.2	Striped Hamming Code . . . . .	36
4.4.3	Hard Disk Storage . . . . .	37
<b>5</b>	<b>Evaluation</b>	<b>38</b>
5.1	Testing Methodology . . . . .	38
5.2	Benchmarks . . . . .	40
5.2.1	Sequential Writes . . . . .	41
5.2.2	Random Word Writes . . . . .	43
5.2.3	Random Page Writes . . . . .	44
5.2.4	Memory Traces . . . . .	45
<b>6</b>	<b>Summary</b>	<b>49</b>
6.1	Conclusions . . . . .	49

6.2 Future Work . . . . .	49
---------------------------	----

# List of Figures

3-1	Properties of the three page states . . . . .	19
4-1	Computational cost of various checksum implementations (2.4Ghz AMD Opteron 150) . . . . .	35
5-1	Checking performance for sequential writes . . . . .	43
5-2	Checking performance for sequential writes using the unoptimized checksum computation . . . . .	43
5-3	Checking performance for random word writes . . . . .	44
5-4	Checking performance for random page writes . . . . .	45
5-5	SPEC CPU2000 Benchmarks Represented In Memory Traces . . . . .	46
5-6	Memory Trace Statistics . . . . .	47
5-7	Checking performance while replaying a 1M entry trace of bzip . . . . .	47
5-8	Checking performance while replaying a 1M entry trace of gcc . . . . .	47
5-9	Checking performance while replaying a 1M entry trace of swim . . . . .	48
5-10	Checking performance while replaying a 1M entry trace of sixpack . . . . .	48

# List of Tables



# Chapter 1

## Introduction

Failing, damaged, or improperly clocked memory can introduce bit errors leading to crashes, freeze-ups, or even data corruption. Even correctly functioning DRAM cells are subject to cosmic rays that can induce transient, or soft errors. Because there are many possible causes of system instability, these problems are notoriously hard to diagnose. This thesis describes **SoftECC**, a software memory testing solution designed to detect and diagnose memory induced stability problems quickly and automatically.

**SoftECC**'s goal is to protect against soft errors in memory without modifying existing applications. As a kernel-level extension to the virtual memory system, **SoftECC** uses the CPU's page-level access permissions to intercept reads and writes before they occur and repeatedly compute page-level checksums. Using the checksums, **SoftECC** verifies that a page's contents are the same at two different points in time. A memory error that happens between two checksum computations will cause the two checksum values to disagree. Legitimate memory writes will also cause a change in checksum value, so a page can only be protected during periods of time when it is not being written to.

## 1.1 Approach

Every pair of consecutive checksums creates an interval of time. At the end of each inter-checksum interval, **SoftECC** can tell three things: if a read has happened, if the page’s contents changed, and if there was a legitimate write. **SoftECC** cannot directly detect errors; however, a page change without a legitimate write implies that an error has occurred. Since writes also imply changes to the page contents, errors are not detectable during inter-checksum intervals containing a legitimate write.

Consider a single page,  $P$ . There are four events of interest that can happen to  $P$ : reads (R), writes (W), checksums (C), and errors (E). The order in which they occur will determine what assurances **SoftECC** can provide about memory integrity. Consider the following possibilities:

Vulnerable Interval: C...W...R...W...R...C  
Detection Interval: C...R...R...R...R...C  
Protection Interval: C.....C

An inter-checksum interval with a legitimate write is a “**vulnerable interval**,” as any errors that occur will go undetected. Two checksum events surrounding a period of time devoid of writes create a “**detection interval**,” a period of time during which any memory errors that occur will be detected. Any inter-checksum interval with neither reads nor writes is called a “**protection interval**,” because **SoftECC** guarantees that any errors that occur on page  $P$  during this time will be detected before the user application has a chance to access invalid data. Furthermore, if **SoftECC** has stored redundancy information for page  $P$ , the error is can be corrected, allowing the user application to continue running.

**SoftECC** creates **detection intervals** inside larger **inter-write intervals**. To detect errors during an inter-write interval, **SoftECC** must calculate a first checksum at the beginning of the interval, and a second checksum before the end of the interval. Because of the cost of these checksums, not all inter-write intervals are worth

protecting. **SoftECC** tries to use checksums to create detection intervals inside the largest inter-write intervals.

Similarly, because **SoftECC** cannot verify the integrity of reads that occur during the same inter-checksum interval as an error, **protection intervals** can only be created inside larger **inter-read intervals**. To reduce an application’s vulnerability to uncorrectable soft errors, **SoftECC** tries to use checksums to create **protection intervals** inside the largest inter-read intervals.

## 1.2 Vulnerability

A page is vulnerable to undetected soft errors during any vulnerable interval (any inter-checksum interval containing a write).

For an application, “**vulnerability**” is defined to be the average number of vulnerable pages (its “**exposure size**”) multiplied by the length of real world time that the computation is running (the “**exposure time**”). The units of vulnerability are page-seconds. **Vulnerability** multiplied by the error density (errors / page / second) yields the expected number of errors that will occur during a computation (which, given that it is a very small number, should be very close to the probability of any errors occurring).

During idle-CPU time, **SoftECC** lowers an application’s vulnerability by checksumming pages to reduce the application’s exposure size. When the CPU is not idle, **SoftECC** can lower soft error vulnerability by reducing an application’s exposure size at the cost of increasing the exposure time.

Creating a **detection interval** of length  $t_{det}$  reduces a computation’s **exposure size** by one page during that interval, decreasing vulnerability by  $t_{det} * 1$  page-seconds. However, any time a page checksum is computed, the user application’s execution is put on hold, increasing the application’s exposure time by one “**checksum time unit**” ( $t_{chk}$ ). Thus, the two checksum operations needed to create a **detection**

**interval** increase vulnerability by  $t_{chk} * n_{vuln}(t_1) + t_{chk} * n_{vuln}(t_2)$  page-seconds, where  $n_{vuln}(t)$  is the number of vulnerable pages at time  $t$ . Note that a decision to checksum a particular interval decreases  $n_{vuln}$  by 1 for all checksums during that interval, and lengthens any encompassing intervals by  $2 * t_{chk}$ .

An omniscient checksumming agent would achieve the minimum possible vulnerability by checksumming any inter-write interval where the length of the interval outweighs the cost of the checksum operations ( $t_{det} > t_{chk} * n_{vuln}(t_1) + t_{chk} * n_{vuln}(t_2)$ ). A realtime checksumming agent, like **SoftECC**, will perform worse than the omniscient case because **SoftECC** cannot accurately predict how long until a page will next be written ( $t_{next}$ ).

At any point in time **SoftECC** can spend one checksum to speculatively begin a **detection interval** worth  $t_{next} * 1$ . However, if **SoftECC** is unlucky and checksums a page shortly preceding an impending write, then the cost of the checksum operations will exceed the value of the **detection interval** for a net *increase* in vulnerability. If **SoftECC** is particularly unlucky, the cost of the second checksum alone will exceed the value of the **detection interval**. In this case, **SoftECC** should allow the write to proceed without checking, giving up the first checksum operation as a sunk cost.

In order to avoid checksumming a page that will soon be written to, **SoftECC** waits to calculate the first checksum until the page has not been written for a period of time. This reduces the chances of wasting time checksumming a page that is about to be written, but it also prevents **SoftECC** from ever detecting over the entire inter-write interval.

## 1.3 Contribution

The main contribution of this thesis is a demonstration of the feasibility of software based memory integrity checking. More specifically, this thesis contributes a simple but effective algorithm for verifying memory integrity, significantly reducing the

chances of receiving undetectable and uncorrectable memory errors. This thesis also contributes an implementation of memory integrity checking in the JOS kernel and an experimental evaluation of the performance characteristics of this implementation. Finally, this thesis contributes suggestions for future work that would allow 100% detection performance and increased error recovery capability.

## 1.4 Rest of Thesis

The remainder of this thesis is structured as follows. Section 3 describes the design of **SoftECC**. Section 4 details the implementation of **SoftECC** as an extension to the JOS kernel. In section 5 we evaluate the performance characteristics of **SoftECC** as it protects several different user-mode access-patterns. Section ?? provides an argument for the utility of **SoftECC** on desktop systems and outlines a method for achieving 100% protection. We conclude in section 6 and provide suggestions for future work.

# Chapter 2

## Background and Related Work

### 2.1 Types of Memory Errors

There are several common causes of memory errors, each with different characteristics.

**Hard Errors** are caused by physical damage to the underlying DRAM circuitry.

Typically they will affect a particular pattern of memory addresses corresponding to a damaged cell, line, or chip. While many types of errors are easily reproducible and will be caught by BIOS memory tests, there are more insidious patterns. For example, if the insulation partially breaks down between several cells, writing to the cells nearby can alter the middle cell's voltage sufficiently to induce a bit flip error. Such errors are pattern dependent and difficult to find. Compounding this difficulty, such problems may occur only when the affected memory chip heats up, or may occur only part of the time.

**Soft Errors** occur when the charge storage representing a bit is sufficiently disturbed to change that bit's value. As process technology advances, both the capacitance and voltage used to store information are decreasing, reducing the "critical charge" necessary to induce a single bit error [9].

**Cosmic Radiation** is a form of soft error that occurs when random high energy

particles (mostly neutrons) impact DRAM cells, disrupting their charge storage. According to Corsair, these single bit errors can happen as often as once a month [6]. And computers in high altitude cities such as Denver face up to 10x the risk of computers operating at sea level [8].

**Package Radio-isotope Decay** can lead to memory errors when trace quantities of radioactive contaminants present in chip packaging decay, emitting alpha particles. There was a particularly infamous problem in 1987 when Po210 from a faulty bottle cleaning machine contaminated an IBM fab producing LSI memory modules. Affected memory chips suffered over 20 times the soft error rate of chips produced at other fabs, leading to a large scale hunt for the contaminant source [7].

**Configuration / Protocol Errors** on a motherboard can cause problems as well. In theory, a DRAM module should perform reliably if the motherboard follows the module's specified timings <sup>1</sup>. However, with so many DRAM and motherboard manufacturers in the marketplace, the potential for incompatibility is great. For example, if the motherboard supplies the DRAM with a lower than specified voltage, the memory will perform slower than it would with its rated voltage. Also, each memory module adds parasitic capacitance to the memory bus. These issues can be a problem for cheap memory modules, which barely meet their specification.

## 2.2 ECC memory: the hardware solution

The problem of DRAM cell reliability is not a new one, and for years manufacturers have been making ECC-DRAM for servers and other machines where reliability is at a premium. ECC-DRAM effectively and transparently protects against single bit DRAM errors and provides detection (without correction) for double bit errors.

---

<sup>1</sup>For current generation memories, this information is stored on the module's EEPROM Serial Presence Detect Chip.

However, ECC-DRAM requires 13% more memory cells per bit (72 cells per 64 bits), and is produced in lower volumes than normal DRAM, both of which increase its price. Although memory pricing is volatile and inconsistent, it would be reasonable to estimate a 33% price premium for ECC-DRAM over comparable non-ECC-DRAM.<sup>2</sup> Thus, while many “high-end” servers use ECC DRAM, most users are rarely even aware of the various memory options and almost never choose to pay the price premium for ECC-DRAM.

No software solution can improve upon the efficacy of hardware ECC at consistency checking DRAM for single bit errors; however with the vast majority of computing devices using non-ECC memory, there is a large space for solutions aiming to close the reliability gap between normal DRAM and ECC-DRAM.

## 2.3 Software Solutions

One software memory testing solution is the Memtest86 utility, which runs an extensive battery of memory test patterns to expose subtle, infrequent, and pattern dependent memory errors missed by power-on BIOS testing [2]. Memtest86 has its own kernel and runs in real mode, requiring a reboot. Because Memtest86 tests can only be performed offline, few users test their memory unless they have reason to suspect it is failing. Furthermore, because it cannot test memory while it is in use, Memtest86 provides no protection against soft errors.

In 2000 Rick van Rein noticed that many memory modules that failed a memtest would fail consistently in the same location [1]. He then wrote a Linux kernel patch called BadRAM that allocates to itself the regions of physical memory known to have problems, preventing faulty DRAM cells from being allocated to user tasks. BadRAM relies on the user to run memtest86 manually to calculate the BadRAM configuration string (passed as a kernel parameter).

---

<sup>2</sup>As of 1/12/05 on Newegg.com: the price range for 1GB of PC3200 DDR DRAM was \$155-\$275 versus \$200-\$408 for ECC-DDR DRAM.



BadRAM is a good approach for isolating hardware faults from the software level. In any system with hardware failure, there will be a delay before a sufficiently competent user or administrator can purchase and install new hardware. Software fault isolation can keep the system up and running while corrective action is taken, allowing hardware to fail-in-place.

# Chapter 3

## Design

**SoftECC** is a software solution for detecting and correcting soft errors in physical memory. As a kernel-level extension to the virtual memory system, **SoftECC**'s operation is transparent to user processes. **SoftECC** defines three page states based on their memory access policies and the status of their stored checksum:

**Hot:** Pages that have been written since the last checksum occurred, and thus do not have a valid checksum stored. User-mode writes and reads proceed normally. If all memory pages are left hot, then **SoftECC** is effectively turned off. Errors that occur while a page is **hot** will not be detected.

**Trapwrite:** Pages that are read-only, and have a valid checksum of their contents stored so that they can be checked for memory errors. **Trapwrite** pages might also have valid redundancy information stored. Writes to **trapwrite** pages must be trapped, so that the page can be checked before the write occurs. Reads can proceed without interruption. In order to minimize the **latency of detection**, **trapwrite** pages should be periodically checked for errors.

**Trapall:** Pages that allow no access, trapping both reads and writes. Because no access is possible, **trapall** pages need not be periodically checksummed.

The table in figure 3-1 shows an overview of the properties of the three page states, which are discussed in more detail in the next few sections.

	Hot	Trapwrite	Trapall
<b>Valid Checksum</b>	no	yes	yes
<b>Valid Redundancy</b>	no	maybe	yes
<b>Writes</b>	poll	trap	trap
<b>Reads</b>	poll	poll	trap
<b>Vulnerable</b>	yes	no	no
<b>Detection</b>	no	yes	yes
<b>Protection</b>	no	maybe	yes

Figure 3-1: Properties of the three page states

### 3.1 State Transitions

In order to keep **SoftECC**'s operation transparent to the user program, the transitions from **trapall** to **trapwrite** to **hot** must be triggered automatically whenever the user program attempts an access operation not allowed by the current state of the page accessed. In other words, writes to any page force it to transition to the **hot** state, and reads to a **trapall** page force it to transition to **trapwrite**. There are no automatic transitions from **hot** to **trapwrite** to **trapall**, and without intervention from **SoftECC**, pages would stay **hot** forever.

The principle of temporal locality says that memory accesses to a particular region tend to be highly clustered in time. In other words, pages that have been accessed recently are likely to be accessed soon and pages that have not been accessed recently are less likely to be accessed soon. Therefore, it makes sense to checksum the **hot** pages that have gone the longest since being written, and the **trapwrite** pages that have gone the longest since being read.

**SoftECC**'s policy with respect with respect to **trapwrite** pages is slightly different. In order to minimize the **latency of detection**, **trapwrite** pages should be checked periodically even if they are still being accessed. However, it is not useful to

continually check pages that are not being accessed. If a **trapwrite** page becomes old enough to be checked again and it has not been accessed, then it will be promoted to **trapall**, stalling its checksum until just before the next pending access.

## 3.2 The Trapwrite State

The **trapwrite** state is for pages that are not expected to be written soon, but might be read. In order to determine if reads are occurring, **SoftECC** periodically checks whether reads have occurred since the last time it checked. If reads don't happen within a certain period of time, **SoftECC** will assume that the page has stopped being read and might not be read for a while. When reads stop occurring on a **trapwrite** page, **SoftECC** promotes it to **trapall**, checksumming it first if reads have occurred since the last checksum.

Even if a **trapwrite** page is still being read and can't be promoted to **trapall**, it should still be periodically checksummed. This decision will validate all the reads that have occurred since the last checksum and prevent any errors that have occurred from affecting future reads.

If both reads and an error have occurred since the last checksum, the program state is highly suspect. Even if the page has stored redundancy information allowing the error to be corrected, the application may have already accessed incorrect data and propagating the error to another location. For this reason, the application should be terminated or reset as gracefully as possible. This may require allowing the application to run with its questionable state just long enough to save any (hopefully intact) user data to disk. A more ambitious approach to error recovery is described in section 6.2.

## 3.3 The Trapall State

The **trapall** state is for pages that are not expected to be read or written soon. While a page is in the **trapall** state, any errors that occur are guaranteed to be detected before they are used. Thus, if these errors are corrected, the user application can continue running. For this reason, **trapall** pages store redundancy information. Redundancy information remains valid until the next write and thus, only needs to be computed during the first **trapall** promotion after each write.

The **trapall** state is also important for performance reasons. On a machine with 1GB of memory, there are a quarter-million physical pages. At any point in time, only a very small fraction of these pages will be in active use. Many of these pages will be used to speculatively cache disk blocks that may never be used. If **SoftECC** had to divide its page checking efforts between all physical pages, there would not be very much compute time to spend on any single page and the level of protection offered would be substantially lower. Instead, the vast majority of pages can be quickly promoted to **trapall** status, allowing **SoftECC** to concentrate its checksumming efforts on the pages that drift in and out of active use.

## 3.4 Design Alternatives

### 3.4.1 The Checksum Queue

Originally, **SoftECC** had a fourth state called **checksum** that would check for writes periodically rather than trapping them. The goal of the **checksum** state was to avoid trapping writes to pages that had been checksummed so recently that computing a second checksum was not worthwhile. After each checksum, a page would spend a short while in the **checksum** queue before being promoted to **trapwrite** (unless a write was detected).

However, the goal of avoiding not-worthwhile second checksum calculations can

also be accomplished by the **trapwrite** state by simply not checking pages that trap early writes. While this method still requires a trap, on a 2.4Ghz Opteron system, checksum computations take 1020ns while traps are under 100ns. Thus, the main cost of trapping a write is the checksum computation, not the actual trap. Because early writes should be rare, it is not clear that marginal benefit of avoiding a rare trap even exceeds the marginal overhead of managing a third page queue.

### 3.4.2 Linux: Swapfile of a modified ramfs partition

One possibility under consideration for achieving memory integrity checking on Linux was to design a modified version of the ramfs filesystem with integrity checking. Ramfs is a simpler version of tmpfs, a special combination block-device/filesystem that stores its contents to virtual memory rather than to a block device. Ramfs is designed to utilize only physical memory and does not support swapping to disk.

The idea was to use a ramfs partition to store a swapfile. The benefit of this approach is that Linux's own memory management system would swap out the least active pages to the modified ramfs disk where they would be checked for integrity.

This approach would not test the design of **SoftECC** in full detail, but would be a means to enable memory protection on a Linux system without having to modify the kernel's memory management system.

Unfortunately, this solution was simply impossible. Linux refuses to use a swapfile stored on either a ramfs or tmpfs partition. This limitation is most likely due to an internal conflict arising from the tight integration of ramfs with the virtual memory system.

# Chapter 4

## Implementation

The JOS kernel was developed to be used as a teaching aid for MIT's Operating Systems class, 6.828. It was designed with simplicity and extensibility in mind, and after being used in the classroom for several years, it is now mature and thoroughly tested. In short it is a ready made platform for testing new operating systems ideas without having to deal with the complexity of a kernel intended for widespread use. The rest of this chapter describes the modifications **SoftECC** makes to the JOS kernel.

### 4.1 Implementing Page State Transitions

**SoftECC** maintains a queue of pages that are waiting to have a periodic action performed. At each timer interrupt (in JOS, this happens 100 times per second), **SoftECC** runs for a while, processing the page queue and checksumming pages. The length of time that **SoftECC** consumes is carefully chosen in order to keep CPU-usage at its target value (see listing 4.1). Each time a page makes its way to the head of the queue, **SoftECC** considers the page (see listing 4.2), taking any necessary actions, and inserting the page back into the tail of the queue.

When a **hot** page reaches the head of the queue, **SoftECC** uses the DIRTY bits

to check whether the page has been written since the last time it was inserted into the queue (see section 4.2.2). If the page hasn't been written to, then **SoftECC** assumes that the user process has stopped writing to the page and promotes it to **trapwrite**. Otherwise, the page is shuffled to the back of the queue to be checked again later.

When a **trapwrite** page reaches the head of the queue, **SoftECC** checks whether the page has been read since the last queue insertion. If it hasn't been read since it was inserted into the queue, **SoftECC** assumes that the user program has stopped reading the page and promotes it to **trapall**. Otherwise, **SoftECC** shuffles the page to the back of the queue, checking whether the page if its checksum age exceeds the threshold.

Listing 4.1: The Periodic Function

```
int      softecc_cpu_load;      // target %CPU load
int32_t  softecc_time_left;     // unused CPU time belonging to SoftECC
uint32_t softecc_time_last;    // the last time SoftECC was run

void on_periodic(void)
{
    uint32_t time_start = get_time();
    uint32_t time_elapsed = (time_start - softecc_time_last);
    softecc_time_left += time_elapsed * softecc_cpu_percent / 100;
    uint32_t time_done = time_start + softecc_time_left;

    while(get_time() < time_done) {
        consider_queue_head(); // do some work
    }

    softecc_time_left -= get_time() - time_start; // may be negative
    softecc_time_last = time_start; // record this start time for next run
}

void on_idle(void)
{
    consider_queue_head(); // do some work
    sys_yield();
}
```



Listing 4.2: The Consider Function

```

PageQueue  page_queue;

void consider(struct Page* pp)
{
    struct Page *pp = remove_head(page_queue);

    switch(pp->state) {
    case HOT:
        if(check_dirty(pp)) {
            // Move to the back of the line
            page_queue.insert_tail(pp);
        } else {
            // Promote to Trapwrite status
            set_checksum(pp);
            pp->state = TRAPWRITE;
            page_queue.insert_tail(pp);
        }
        break;
    case TRAPWRITE:
        if(check_access(pp)) {
            // Do I renew the checksum to minimize checksum-age?
            if(checksum_age(pp) > param_max_checksum_age) {
                check_page(pp);
            }
            // Move to the back of the line
            page_queue.insert_tail(pp);
        } else {
            // Promote to TrapAll status
            if(pp->flags & ACCESS) {
                check_page(pp);
            }
            pp->state = TRAPALL;
        }
        break;
    case TRAPALL:
        shrink_redundancy_information(pp);
        page_queue.insert_tail(pp);
        break;
    }
}

```

#### 4.1.1 Read and Write Trap Handlers

If **SoftECC** traps a write to either a **trapwrite** or **trapall** page (see section 4.2.1), that page must transition to the **hot** state before the write can occur (see listing 4.3).

As discussed in section 1.2, it is not always worthwhile to compute a second checksum. However, as long as the page's checksum-age and thus, the size of the potential inter-checksum interval exceeds the threshold, **SoftECC** will check the page. Next, **SoftECC** inserts the page back into the queue so that it will be periodically checked to determine when the user-mode program is done writing to it. Finally, the time used to perform these steps must be counted against **SoftECC**'s CPU usage for the purposes of load throttling (see section 4.1).

Likewise, if **SoftECC** traps a read to a **trapall** page, that page must transition to the **trapwrite** state before the write can occur. As with writes, the second checksum is only performed if the value of the potential protection interval exceeds a threshold. Note that, if the user-mode application issues a write just after the read (which is common), then the page will fail the threshold test and transition directly to the **hot** state for only the cost of the trap.

Listing 4.3: Read and write trap handlers

```

void trap_write(struct Page* pp)
{
    time_t time_start = get_time();

    if(checksum_age(pp) >= param_min_detect_threshold()) {
        check_page(pp);
    }
    pp->state = HOT;
    page_queue.insert_tail(pp);

    softecc_time_left -= (get_time() - time_start);
}

void trap_read(struct Page* pp)
{
    time_t time_start = get_time();

    if(checksum_age(pp) >= param_min_protect_threshold) {
        check_page(pp);
    }
    pp->state = TRAPWRITE;
    page_queue.insert_tail(pp);

    softecc_time_left -= (get_time() - time_start);
}

```

## 4.2 The JOSkern VMM extensions

The JOS kernel VMM is fairly simple, relying primarily on the Pagestructure to represent physical pages (see listing 4.4). The original Page structure is fairly spartan, containing only a reference count, and a free list pointer.

**SoftECC** extends the Pagestructure with five items. Each page needs to store its current state number and most recent checksum for consistency checking. In order to implement the page queue (see section 4.1), another link was required as well a variable to store the time when the page was inserted into the queue (to determine queue age; see section 4.1). Because trapping accesses to a physical pages requires modifying all the virtual mappings that point to it, each page has a pointer to a list

of virtual mappings (see section 4.2.1). Lastly, **SoftECC** adds eight bits worth of flags to the Page structure. Two of these will be needed for the special access and dirty bits (see section 4.2.2).

Listing 4.4: The Page Structure

```
struct Page {
    Page *    pp_link;        // free list link
    uint16_t  pp_ref;        // reference count

    uint8_t   state;         // new: current page state
    checksum_t checksum;     // new: stored checksum value
    Page *    queue_link;    // new: page queue link
    uint32_t  queue_time;    // new: time of last queue insertion
    Mapping * va_map_list;   // new: list of mapped va entries
    uint8_t   flags;        // new: flags, esp accessed and dirty
};

struct Mapping {
    Mapping * next_link;    // linked-list pointer
    pte_t *   pte;         // address of PTE that maps to Page
};
```

### 4.2.1 Trapping Memory Access on x86

Removing the write (PTE\_W) and user (PTE\_U) permission bits from a PTE causes subsequent user-mode writes and reads to the corresponding virtual page to throw a page fault which **SoftECC** can intercept. In order to trap accesses to a physical page, **SoftECC** needs to modify all PTE's that map to that page.

Because **SoftECC** removes the PTE\_W bit to enable write trapping, it needs to know the original state of that bit in order to restore the bit during trapped write. In other words, **SoftECC** needs a way to distinguish between read-only virtual mappings to **trapwrite** (and **trapall**) physical pages and originally writable virtual mappings

to **trapwrite** pages. Because a physical page can have multiple virtual mappings with differing permissions, this information must be for each virtual mapping. Because user-mode pages in JOS always have the PTE\_U bit set, its original status is unambiguous.

Intel's x86 architecture defines three PTE bits to be available for operating system use. Of these, JOS kernel utilizes two, referred to as PTE\_COW (Copy On Write) and PTE\_SHARE, and are both used primarily in the implementation of fork. When a process forks, JOS marks all writable pages as copy on write and clears the PTE\_W bit. When the next user-mode write is trapped, the page is remapped to a copy of the original, preventing writes from one process becoming visible in the other. However, pages with the PTE\_SHARE bit enabled are mapped directly by fork so that writes will be visible to both processes, allowing inter-process communication. This leaves only one available PTE bit remaining.

**SoftECC** defines the last remaining PTE bit to be PTE\_HOW (Heat On Write). When **SoftECC** clears the PTE\_W bit, it stores the original state of that bit to PTE\_HOW, and when trapping a write restores PTE\_W only if PTE\_HOW is set.

When a page is trapping writes (the page state is either **trapwrite** or **trapall**), PTE\_HOW is a proxy to the “real” value of PTE\_W as concerns the rest of the kernel. Thus, any statement that wants to read PTE\_W should instead read the logical “or” of both PTE\_W and PTE\_HOW. Any statement that would have written PTE\_W should now write PTE\_HOW if the underlying page is trapping writes and PTE\_W if not. These changes can also be abstracted by using the accessor methods defined in listing 4.5. For example, when the original version of fork sets PTE\_COW, it clears PTE\_W. Thus fork now clears both PTE\_W and PTE\_HOW.

Because JOS is an exokernel design [11], many of the more complicated operating system features are implemented in user-space and call into `sys_page_map` to change permission bits. Thanks to the flexibility of the exokernel design, updating this system call was sufficient to ensure correctness for the majority of JOS kernel's feature set.

Listing 4.5: PTE\_HOW Compatible Accessor Methods

```

bool get_PTE_W (pte_t pte)
{
    return (pte & PTE_W) || (pte & PTE_HOW)
}
void set_PTE_W (pte_t* pte)
{
    struct Page* pp = pte2page(pte);
    if(pp->state == TRAPWRITE ||
        pp->state == TRAPALL)
        *pte |= PTE_HOW; // set PTE_HOW
    else
        *pte |= PTE_W;   // set PTE_W
}
void clear_PTE_W (pte_t* pte)
{
    *pte &= ~PTE_W;     // clear PTE_W
    *pte &= ~PTE_HOW;  // clear PTE_HOW
}

```

#### 4.2.2 Checking for Prior Memory Access on x86

To check for prior memory accesses to a page, **SoftECC** can inspect the accessed (PTE\_A) and dirty (PTE\_D) PTE bits. These bits are set automatically by the CPU whenever the corresponding virtual page is read from or written to. A physical page should be considered accessed or dirty if any of the PTE's where it is mapped are marked accessed or dirty. Periodically polling the accessed and dirty bits does not allow **SoftECC** to intercept page access events before they occur; however, polling avoids the overhead of trapping into kernel space.

There is one additional complication to consider when polling. If one of a page's virtual mappings is accessed and then unmapped, the corresponding PTE will no longer be associated with the physical page and thus the accessed and dirty bits will be lost. This could result in a dirty physical page appearing to be not dirty. For this reason, whenever a PTE is about to be overwritten, the status of its accessed and dirty bits needs to be retained. This is accomplished by "or"ing these bits with special per physical page ACCESS and DIRTY bits (stored in the Page structure; see listing 4.4).

Listing 4.6: Checksum accessor methods

```
void check_page(struct Page* pp)
{
    if( calculate_checksum(pp) != pp->checksum) {
        handle_memory_error(pp);
    }
    clear_access(pp); // Reset ACCESS bits
    clear_dirty(pp); // Reset DIRTY bits
}

void set_checksum(struct Page* pp)
{
    pp->checksum = calculate_checksum(pp);
    clear_access(pp); // Reset ACCESS bits
    clear_dirty(pp); // Reset DIRTY bits
}
```

## 4.3 Checksums

The simplest and quickest x86 checksum computation is a dword length (32 bit) xor operation performed over the entire 4k page (see listing 4.7). Effectively, this produces 32 individual parity bits, each one covering a stripe of 1024 bits.

Since each stripe only has single bit parity, some double-bit errors are not detectable; however, this limitation is not a problem. The probability of receiving a true, instantaneous double bit error<sup>1</sup> is exceptionally low [7]. And the probability of receiving two single-bit errors within the same 1024 bit stripe within the same inter-checksum interval is very low as well. The probability of a double-bit error within a single stripe during an period of time is the square of the probability of a single-bit error during the same period of time. Unless the inter-checksum interval extends so long (multiple years) that the probability of a single-bit error (within a single 1024 bit stripe) rises significantly, double-bit errors will be exceedingly rare, and detection performance will be well defined by the vulnerability metric (see section 1.2).

Listing 4.7: The Checksum Computation

```
checksum_t calculate_checksum_c(struct Page* pp)
{
    // get a pointer to page contents
    checksum_t* p = (checksum_t*)page2kva(pp);
    checksum_t ret = 0;
    int i;
    for(i=0; i<1024; i++) {
        ret ^= p[i]; // xor
    }

    return ret;
}
```

Unfortunately, the default assembly code for checksum calculation generated by gcc (see listing 4.8) is less than optimal, requiring almost 3000 ns to execute on a

---

<sup>1</sup>Actually, double-bit errors in DRAM need not be caused by the exact same event (eg high energy cosmic neutron). Two single-bit errors within the same refresh cycle (approx 4-64ms, barring earlier programmatic access) is sufficient.



2.4Ghz AMD Opteron 150 system with 1GB of DDR400 memory (see the table in figure 4-1). The core loop is a single implicit memory load and a trivial xor instruction; however, each loop iteration executes three instructions for loop control.

Far less obvious, there is a subtle alignment problem with the compiler generated code (see listing 4.9). The core loop crosses a 16 byte boundary, putting it in a different two different cache lines. The loader aligns the start of the function to a 16 byte boundary, but the compiler is not smart enough to realize just how important the alignment of the core loop is. Simply inserting a few nop's before the core loop improves performance to 2000 ns.

Forcing the compiler to unroll the loop provides considerable benefit (see listing 4.10). The core loop is now 16 instructions, effectively amortizing the overhead of the loop control instructions. Checksum now runs in 1489 ns.

Utilizing the 128bit packed xor from Intel's MMX instruction set allows for even more improvement, bringing execution time to 1058 ns (see listing 4.11). The core loop is now a scant 8 pxor instructions that do the work of 32 regular xors. With only 8 instructions consuming 128 bytes of data, the calculation is now heavily bound by memory bandwidth.

Finally, it is possible to optimize the memory access just slightly by interleaving the instructions to exercise multiple cache lines at once, checksumming an entire page in just 1024 ns (see listing 4.12).

Listing 4.8: Compiler-generated (gcc -O3) assembly code for calculate\_checksum

```
calculate_checksum:
{...}
xor    %eax, %eax           ; set edx = p (start of page)
xor    %ecx, %ecx           ; set eax = 0
xor    %ecx, %ecx           ; set ecx = 0
loop:
xor    (%edx, %ecx, 4), %eax ; eax = eax xor32 MEM[edx + 4*ecx]
inc    %ecx                ; increment index
cmp    $1024, %ecx          ; compare
jle   loop                 ; loop if ecx < 1024
{...}                       ; return eax
```

Listing 4.9: Annotated objdump output for compiler-generated code

```

080485d0 <calculate_checksum_c>:
-----16-BYTE-BOUNDARY-----
80485d0:    55                push   %ebp
80485d1:    89 c5             mov    %esp,%ebp
80485d3:    8b 4d 08          mov    0x8(%ebp),%ecx
80485d6:    31 c0             xor    %eax,%eax
80485d8:    31 d2             xor    %edx,%edx
80485da:    89 f6             mov    %esi,%esi
core_loop_start:
80485dc:    33 04 91          xor    (%ecx,%edx,4),%eax
80485df:    42                inc   %edx
-----16-BYTE-BOUNDARY-----
80485e0:    81 fa ff 03 00 00  cmp    $0x3ff,%edx
80485e6:    7e f4             jle   80485dc <core_loop_start>
80485e8:    c9                leave
80485e9:    c3                ret

```

Listing 4.10: Loop-unrolled (gcc -O3 -funroll-loops) assembly code for calculate\_checksum

```

calculate_checksum:
{...}                ; set edx = p (start of page)
xor   %eax,%eax       ; set eax = 0
xor   %ecx,%ecx       ; set ecx = 0
loop:
xor   0x00(%edx,%ecx,4),%eax ; eax = eax xor32 MEM[edx + 4*ecx + 0]
xor   0x04(%edx,%ecx,4),%eax ; eax = eax xor32 MEM[edx + 4*ecx + 4]
...
xor   0x3c(%edx,%ecx,4),%eax ; eax = eax xor32 MEM[edx + 4*ecx + 60]
addl $0x10,%ecx       ; increment index
cmp   $1024,%ecx      ; compare
jle  loop             ; loop if ecx < 1024
{...}                ; return eax

```

Listing 4.11: MMX Loop-unrolled assembly code for calculate\_checksum

```

calculate_checksum:
{...}                ; set edx = p (start of page)
pxor  %xmm1,%xmm1     ; set xmm1 = 0
xor   %ecx,%ecx       ; set ecx = 0
loop:
pxor  0x00(%edx,%ecx),%xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 0]
pxor  0x10(%edx,%ecx),%xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 16]
...
pxor  0x70(%edx,%ecx),%xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 112]
addl $0x80,%ecx       ; increment index
cmp   $4096,%ecx      ; compare
jle  loop             ; loop if ecx < 4096

subl $0x10,%esp       ; allocate 128 bits of stack space
movdqu %xmm1,(%esp)   ; and store xmm1
xor   %eax,%eax       ; eax = 0
xor   0x00(%esp),%eax  ; eax = eax xor xmm1[0:31]
xor   0x04(%esp),%eax  ; eax = eax xor xmm1[32:63]
xor   0x08(%esp),%eax  ; eax = eax xor xmm1[64:95]
xor   0x0c(%esp),%eax  ; eax = eax xor xmm1[96:127]
addl $0x10,%esp       ; release stack space
{...}                ; return eax

```

## 4.4 Redundancy for Error Correction

To enable error correction, enough redundant information must be stored about the contents of a page that the location of a single-bit error is determinable. There are at least three options, each with various advantages:

Listing 4.12: Interleaved MMX Loop-unrolled assembly code for calculate\_checksum

```

calculate_checksum:
    {...}                ; set edx = p (start of page)
    pxor  %xmm1, %xmm1    ; set xmm1 = 0
    xor   %ecx, %ecx     ; set ecx = 0
loop:
    pxor  0x00(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 0]
    pxor  0x40(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 64]
    pxor  0x10(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 16]
    pxor  0x50(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 80]
    pxor  0x20(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 32]
    pxor  0x60(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 96]
    pxor  0x30(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 48]
    pxor  0x70(%edx, %ecx), %xmm1 ; xmm1 = xmm1 xor128 MEM[edx + ecx + 112]
    addl  $0x80, %ecx     ; increment index
    cmp   $4096, %ecx    ; compare
    jle  loop           ; loop if ecx < 4096

    {...}                ; eax = xor(xmm1[0:31], xmm1[32:63], xmm1[64:95], xmm1[96:127])
    {...}                ; return eax
    
```

Checksum	Cost
Plain C	2932 ns
Aligned C	2000 ns
Loop-unrolled 32-bit	1489 ns
Loop-unrolled MMX	1058 ns
Interleaved-unrolled MMX	1024 ns

Figure 4-1: Computational cost of various checksum implementations (2.4Ghz AMD Opteron 150)

### 4.4.1 Full Memory Copy

The simplest redundancy scheme is to allocate a second page, and make a full duplicate copy of the original. The drawback to this approach is poor storage efficiency, requiring double the physical DRAM. However, this method has a compelling advantage: it is very fast (1694 ns).

Like the normal checksum operation, a memory copy is bounded by the DRAM bandwidth, except that whereas a checksum condenses the incoming data to a single dword, a memory copy writes a full page back to DRAM. Because both the checksum and memcpy operations require reading the same data from memory, calculating a checksum while performing a memcpy takes no more time than performing the memcpy alone.

### 4.4.2 Striped Hamming Code

A hamming code[12] is perhaps the most compelling of the traditional error correcting coding schemes. Most importantly, it does not require mangling the original data bits. Also, it is reasonably compact and only moderately computationally expensive.

Hamming coding works by setting the value of every bit occupying a power of two address in the signal (eg, the 1st bit, 2nd bit, 4th bit, 8th bit...  $2^N$  bit) to be the xor of all bits whose addresses' binary representations have the Nth bit set. These bits are the parity bits. The bits not located at power of two addresses are data bits. Naturally, in the actual implementation, the logical Hamming code signal addresses will not match the actual addresses of the bits. The data bits will remain in place, while the parity bits will be stored in an array of redundancy data.

To correct a single-bit error, simply flip the bit at the address that is the sum of the addresses of all the parity bits that are incorrect. If only one parity bit is incorrect, the error is that parity bit. In order that errors to the parity bits be detected, the pages containing parity bits are checksummed by **SoftECC** in the same manner as

normal pages, except that because only **SoftECC** accesses these pages, they need not be trapped or polled like normal pages.

### 4.4.3 Hard Disk Storage

The third redundancy level is to store a duplicate of the page's contents to disk. This has the advantage of requiring no extra memory storage; however, disk access is quite slow. Even the fastest hard disks still have access times of several milliseconds, thousands of times longer than  $t_{chk}$ . Not all of this time requires CPU attention, but the CPU-overhead of disk management will still be greater than for the other two schemes.

In some cases, hard disk redundancy already exists. The executable images and the file cache originated from copies on disk. Also, the virtual memory system on commercial operating systems periodically copies dirty pages to a swapfile in case they need to be swapped out to make room for other memory uses.

# Chapter 5

## Evaluation

### 5.1 Testing Methodology

In order to analyze the performance of **SoftECC** on various workloads, a test harness was created. **SoftECC**'s checking was disabled for the test harness, and only the memory pages for the benchmark applications were included in the results.

The first plan for testing was to introduce errors at random memory locations at randomized randomized time intervals and count how many were detected. Because of the cyclical nature of the benchmarks, each page is guaranteed to be accessed at least once per iteration, so any errors not detected after a complete iteration were assumed to be missed. Because the results of the user-mode benchmarks are not of interest, errors need not be corrected as they will not affect the program flow. Unfortunately this conservative benchmarking strategy was extremely slow, taking hours to produce a single data point.

An important properties of soft errors are that they are randomly distributed in both space and time. Put another way, the probability or density of soft errors is evenly distributed in space and time. Rather than introduce errors one at a time randomly distributed between the various pages and averaging the results to achieve a probability, introducing one error in each page would directly measure the instan-

taneous probability of detecting a randomly placed error.

Furthermore, the instantaneous probability of error detection or correction can be measured without actually introducing errors and waiting. Errors that occur on a page during a vulnerable interval will never be detected. Errors during either detection or protection intervals will always be detected. Thus the instantaneous probability of detection is simply the fraction of pages that are currently in a detection or protection intervals.

Because all **hot** pages have been written at least once and do not have valid checksums, **hot** pages are always in vulnerable intervals. Because they trap reads and are guaranteed to have been checksummed since the last access, **trapall** pages are always in protection intervals. Because they do not trap reads, the correct interval type of a **trapwrite** is not known until it is next checksummed. If a **trapwrite** page is checksummed without being accessed since its last checksum, then it just completed a protection interval. Otherwise, it was a detection interval.<sup>1</sup>

Because pages only change inter-checksum interval status when they are checksummed, it is possible to do better than randomly sampling the instantaneous probability of detection. By using the vulnerability metric, the overall probability of detection for an interval of time can be calculated by monitoring for page checksum events (see listing 5.1).

While this improved methodology does not provide the same tangibility as introducing and detecting actual errors, it is not only much much faster; it is also more accurate, directly measuring the quantities of interest rather than approximating them through statistical methods.

---

<sup>1</sup>If **trapwrite** pages are accessed while their checksum is still young, then they will transition to **hot** without checksumming first, rendering the entire interval vulnerable. This design is fine, however, because the next checksum will occur when the page is **hot**, correctly counting the time the page spent in the **trapwrite** state as being vulnerable to silent errors.

Listing 5.1: Statistics gathering for inter-checksum intervals (at checksum events)

```

void stats_on_checksum(struct Page* pp)
{
    pause_time();

    // Calculate the length of the Inter-Checksum Interval
    time_t ICI_duration = get_time() - pp->checksum_time;
    pp->checksum_time = get_time();

    switch(pp->status) {
        case HOT:
            // By design hot pages have been written at least once
            // Thus, this was a vulnerable interval
            stats.vuln += ICI_duration;
            break;

        case TRAPWRITE:
            // This is the complex one...
            // By design there were no writes, but maybe reads, so
            // Check the ACCESS bit(s)
            check_access(pp);
            if(pp->flags & ACCESS)
                stats.detect += ICI_duration;
            else
                stats.protect += ICI_duration;
            break;

        case TRAPALL:
            // By design, there were no reads, no writes:
            // Thus, this was a protection interval
            stats.protect += ICI_duration;
            break;

    }

    unpause_time();
}

```

## 5.2 Benchmarks

Perhaps the most important consideration when benchmarking **SoftECC** is to decide what user program to execute. The memory access patterns of the user program will have a far greater impact upon the performance of **SoftECC** than any other single factor. Unfortunately, many traditional benchmarks end up as pathological best or



worst cases.

For example, many CPU intensive benchmarks have a very small memory footprint and operate primarily on a single page of stack memory. For a program with such a small memory footprint, **SoftECC** would be unable to significantly impact reliability. At any point in time the program's error exposure will already be limited to its single page of memory. And if **SoftECC** were to checksum this memory page, it would almost immediately be written to.

What follows are preliminary benchmarks that outline some of the factors affecting **SoftECC**'s performance.

### 5.2.1 Sequential Writes

Consider a benchmark that repeatedly performs sequential writes to memory (see listing 5.2). Such a memory access pattern has bad temporal locality because the pages most recently accessed have the least chance of being accessed, while the oldest pages are most likely to be accessed next. Because **SoftECC** uses a LRU-like scheme, it will tend to checksum the oldest pages first, just as they are about to be written to again.

Listing 5.2: Sequential Write Benchmark

```
int array_size = 1000 * (PGSIZE/4);
uint32_t array[array_size];

void bench_seq_write() {
    while(1) {
        for(int i=0; i<array_size; i++) {
            array[i] = i;
        }
    }
}
```

Unless **SoftECC** can checksum pages faster than they are written, it will spend all of its time working on checksumming pages that will soon be overwritten. This is

demonstrated in figure 5-1, which shows **SoftECC**'s detection rate hovering near zero until the checksumming exceeds the rate of writing around 25% CPU-load. After this point, **SoftECC** is able to checksum pages immediately after the user application finishes writing to them and detection performance rises to nearly 100%.

At 25% CPU-load (33% CPU-overhead), **SoftECC** has a 0.33:1 CPU usage ratio with the user application. This indicates that the user application takes 3 times as long to write a page as **SoftECC** takes to checksum a page. Herein lies the significance of optimizing **SoftECC**'s checksum algorithm (see section 4.3). Using the unoptimized checksum algorithm, **SoftECC** the rise in detection doesn't occur until 45% CPU-load (81% CPU-overhead) (see figure 5-2).

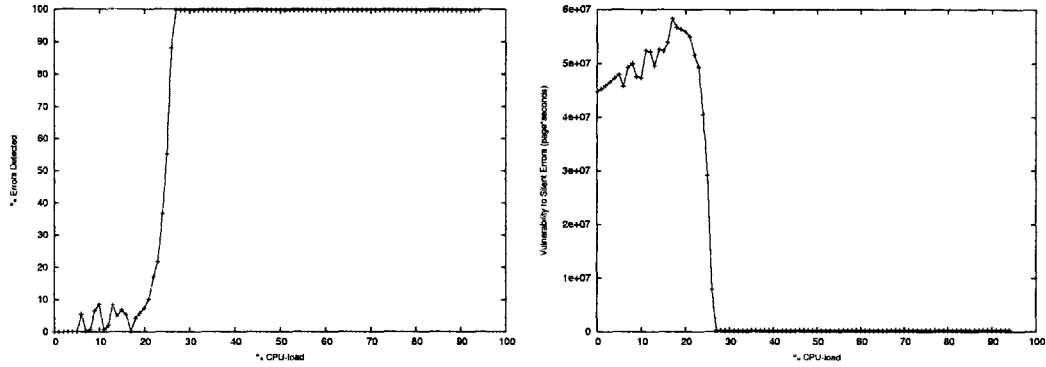


Figure 5-1: Checking performance for sequential writes

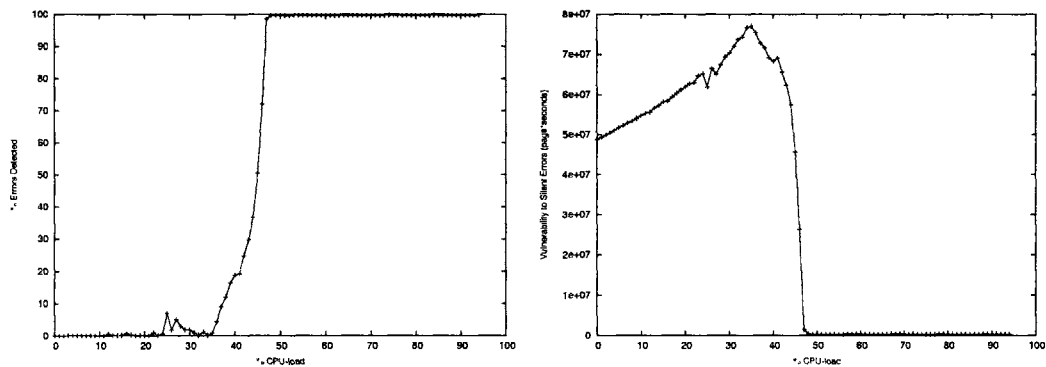


Figure 5-2: Checking performance for sequential writes using the unoptimized checksum computation

## 5.2.2 Random Word Writes

Consider a benchmark that allocated all available memory to a single array that was written with random accesses (see listing 5.3). Because this benchmark exhibits no temporal locality, we expect **SoftECC** to perform poorly.

As can be seen in the left graph of figure 5-3, **SoftECC** doesn't manage to protect a significant fraction of the data pages until its checksumming keeps up with the rate of page turnover, allowing it to compute a checksum after each memory write. For random word writes, this implies that **SoftECC** will need an order of magnitude more compute time than the user-mode code <sup>2</sup>. Because of this, **SoftECC**'s failed

<sup>2</sup>It takes 1024 dword reads to checksum a page, versus 1 dword write to invalidate that checksum.

Listing 5.3: Random Word Write Benchmark

```

int array_size = 1000 * (PGSIZE/4);
uint32_t array[array_size];

void bench_rand_word_write() {
    while(1) {
        uint32_t r = sys_genrand();
        int i = r % array_size;
        array[i] = r;
    }
}

```

efforts increase the **exposure time** without decreasing the **exposure size**, yielding a net *increase* in the vulnerability metrics that only gets worse with increasing CPU use (see the right graph in figure 5-3).

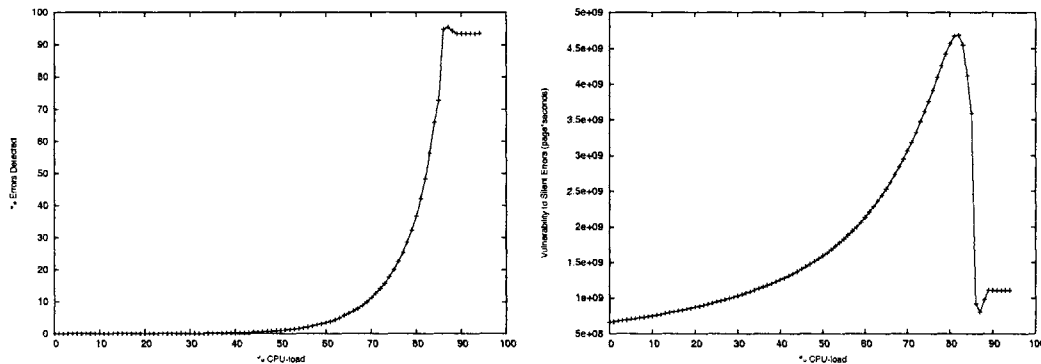


Figure 5-3: Checking performance for random word writes

### 5.2.3 Random Page Writes

While protecting fine-grained random word writes is costly, the situation is quite different for *coarse* grained random writes. Consider a benchmark that simulates the activity of a block cache for a filesystem by picking a random page and writing to it (see listing 5.4).

---

The only thing that prevents the rise in this graph from occurring at 99.9% CPU-load is that random number calculations are expensive, dominating the runtime of the user-code.

Listing 5.4: Random Page Write Benchmark

```

int array_size = 1000 * (PGSIZE/4);
uint32_t array[array_size];

void bench_rand_page() {
    while(1) {
        uint32_t r = sys_genrand();
        int i = r % 1000;
        for(int j=0; j<1024; j++) {
            array[i*PGSIZE+j] = r;
        }
    }
}

```

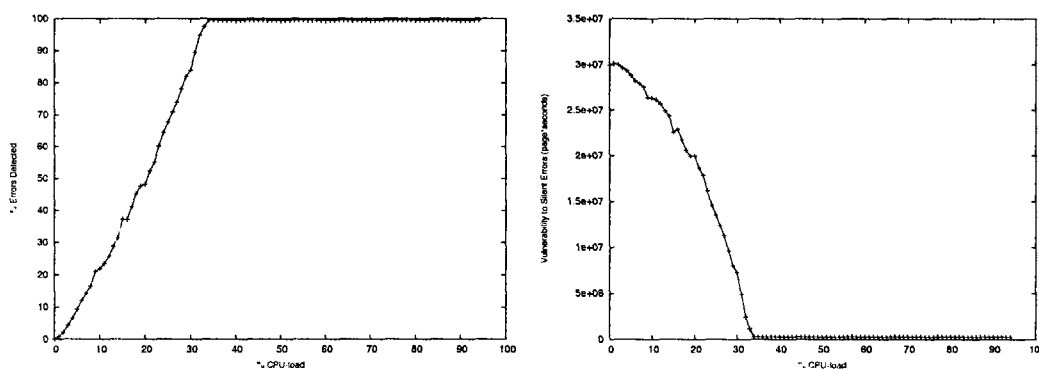


Figure 5-4: Checking performance for random page writes

## 5.2.4 Memory Traces

While artificial benchmarks can provide many insights as to the factors affecting **SoftECC**'s performance, ultimately, it is **SoftECC**'s integrity checking performance on real-world applications that is of greatest interest. Real-world applications have far more complexity than can be captured by an artificial benchmark. Unfortunately, it is not feasible to port a large application such as gcc to JOS kernel, as this would require emulating a significant fraction of the standard POSIX kernel API calls. For this reason, we acquired 1 million entry memory access traces from four applications: bzip, gcc, and swim. These traces were published as a part of the course materials for Notre Dame's Operating Systems Principles class, CSE 341 [10], and were derived from four SPEC CPU2000 benchmarks of the same names (see table 5-5) [13].

Benchmark	Category	Suite
176.gcc	C Programming Language Compiler	CINT2000
256.bzip2	Compression	CINT2000
171.swim	Shallow Water Modeling	CFP2000
200.sixtrack	High Energy Nuclear Physics Accelerator Design	CFP2000

Figure 5-5: SPEC CPU2000 Benchmarks Represented In Memory Traces

Figures 5-7, 5-8, 5-9, and 5-10 reflect the performance of **SoftECC** while replaying 1 million entry traces of bzip, gcc, swim, and sixpack. One striking feature of these graphs is that on all four benchmarks, **SoftECC** is able to achieve approximately a 50% error detection rate with less than 1% CPU-load, which implies that about half of the pages utilized by these applications are being accessed in a read-only manner. This observation is confirmed by the statistics in table 5-6.

There is no similar effect for the correction rate, implying that every page is being accessed repeatedly. Unfortunately, the source of these repeated accesses is the looping of the memory trace.

As demonstrated in table 5-6, for gcc, swim, and sixpack, there are a significant number of pages that are only accessed once during the trace. However, these accesses occur each time the memory trace loops. Because the memory traces are so short, these accesses appear periodic to **SoftECC**, much like the access patterns in the sequential write benchmark. Consequently, the number of single access pages is roughly proportional to the overhead threshold where protection performance begins to rise. Similarly, the number of single write pages is roughly proportional to the overhead threshold where detection performance begins to rise. This suggests that these memory traces are insufficiently short to demonstrate **SoftECC**'s performance, and better benchmarks are needed.

Trace	bzip	gcc	swim	sixpack
Writes	122419	107184	67170	160983
Reads	877581	892816	932830	839017
Unique Addresses	11113	37697	93694	84920
Unique Pages	317	2852	2543	3890
Read-only Pages	167	1707	1374	1825
Pages Written 1 time	26	471	397	827
Pages Written 2 times	10	148	124	271
Pages Accessed 1 time	16	886	690	1399
Pages Accessed 2 times	19	293	292	500

Figure 5-6: Memory Trace Statistics

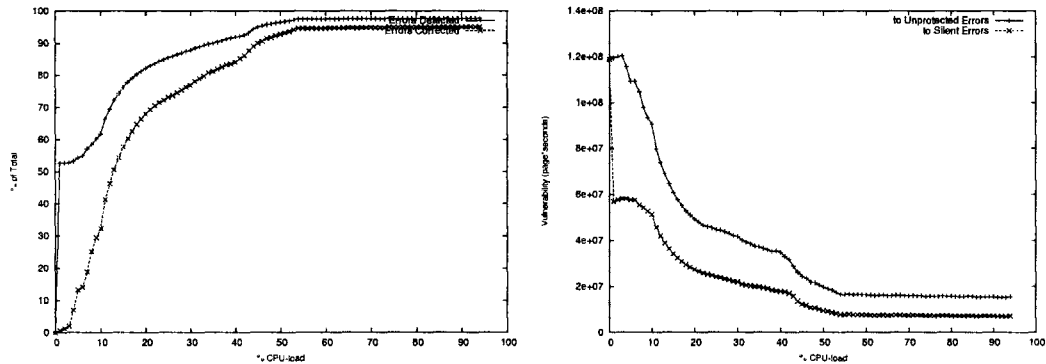


Figure 5-7: Checking performance while replaying a 1M entry trace of bzip

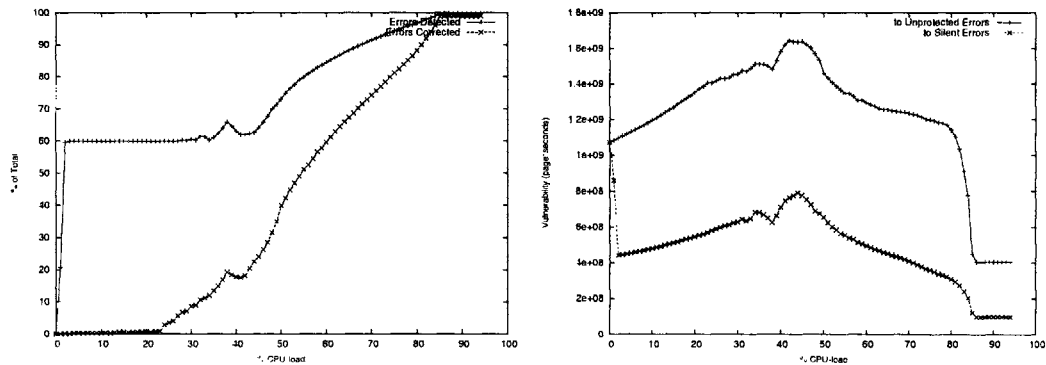


Figure 5-8: Checking performance while replaying a 1M entry trace of gcc

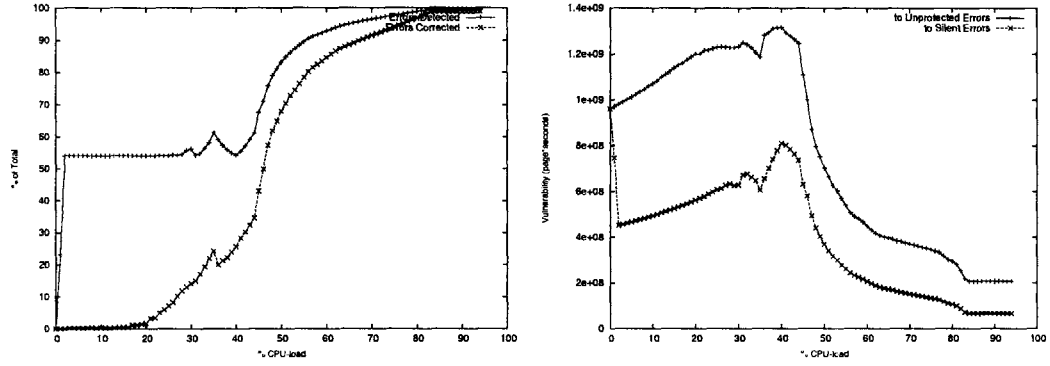


Figure 5-9: Checking performance while replaying a 1M entry trace of swim

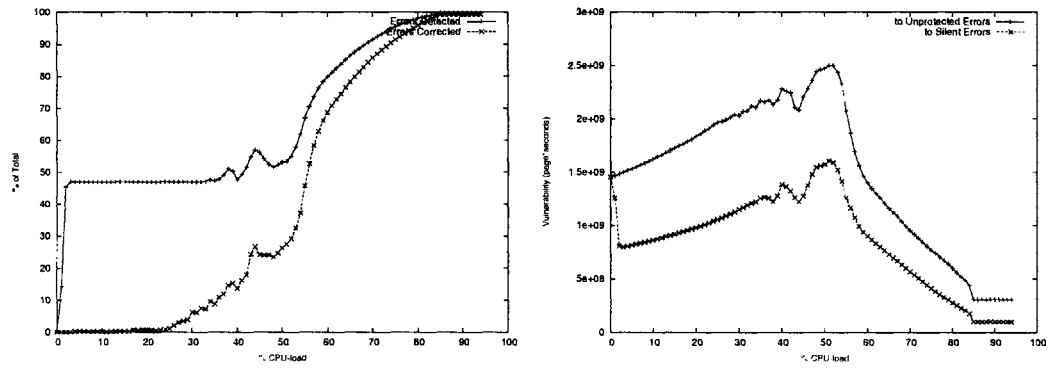


Figure 5-10: Checking performance while replaying a 1M entry trace of sixpack



# Chapter 6

## Summary

### 6.1 Conclusions

The preliminary benchmarks indicate that **SoftECC** can halve the number of undetected soft error using only minimal compute time. Most multi-user, multi-process operating systems in use today exhibit significant spacial and temporal locality of data access and stand to benefit greatly from the added reliability **SoftECC** can provide. Because it is implemented at the kernel-level, **SoftECC**'s operation is transparent to user-mode applications. This indicates that for a minimal overhead cost, **SoftECC** can provide added protection against soft errors to existing systems. Furthermore, **SoftECC** is capable of exploiting idle CPU-time to perform its checks.

### 6.2 Future Work

#### Linux Implementation

Ideally, **SoftECC** would have been implemented as a patch to the Linux kernel. However, there are significant implementation issues to overcome before **SoftECC** on Linux can become a reality. Perhaps the greatest challenge is to find ways to

minimize **SoftECC** 's impact on other virtual memory features. For x86 systems, the only practical way to trap user mode memory accesses is to modify the virtual page table entries by removing the Write, Present, or User permission bits. Overloading the functionality of the page table permission bits requires checking every statement within the kernel that references these bits in order to verify that existing kernel functionality has not been compromised.

While the size and complexity of the Linux kernel prevented a Linux implementation during this iteration of the project, an implementation on a full-featured OS will be necessary before **SoftECC** can have any broad applicability.

### **Enhanced Error Recovery**

The recovery scenario to a correctable but not protected (detected before potential use) error would be much better if it were possible to emulate the backward execution of the application code. In some (but clearly not all) cases, this can be done even on x86. In the best case, an error could be demonstrably unused, and the application could continue running. In other cases, it is possible to prove that the error propagated to certain other memory locations, which could also be corrected by emulating the reexecution of the instructions that used tainted data. Even if it is not possible to prove that all tainted memory locations had been fixed, the application level recovery scheme (graceful termination) would have a better chance of success.

### **Improved Benchmark Results**

Unfortunately, due to constraints on what user-mode benchmarks could be ported to the JOS kernel, the benchmarking results are incomplete. Future work entails acquiring or recording more extensive memory access traces, including access timing information. These would be used to give a much clearer picture of **SoftECC**'s performance in real-world scenarios. Alternatively, real-world applications could be ported to run on top of the JOS kernel.

# Bibliography

- [1] van Rein R. (2000). **BadRAM: Linux kernel support for broken RAM modules** <http://rick.vanrein.org/linux/badram/>
- [2] Brady C. (1999 - 2004). **Memtest86 3.2 (GPL)** <http://www.memtest86.com/>
- [3] Demeulemeester S. (2004). **Memtest86+ 1.50 (GPL)** <http://www.memtest.org/>
- [4] Cazabon C. (1999). **Memtester 4.0.5 (GPL 2)** <http://pyropus.ca/software/memtester/>
- [5] Harbaugh T. (2005). **Blue Smoke devel-20050509** ( <http://bluesmoke.sourceforge.net/>
- [6] Lieberman D, VP Engineering. (1998). **ECC Whitepaper**. CorsairMicro. July 30, 1998. <http://www.corsairmicro.com/main/tecc.html>
- [7] Ziegler JF. (1994). **IBM experiments in soft fails in computer electronics (1978-1994)** IBM Journal of Research and Development. Vol 40 No 1 1996. <http://www.research.ibm.com/journal/rd/401/tocpdf.html>
- [8] Wilson R, Lammers D. (2004). **Soft errors become hard truth for logic** EE Times Online. May 3 2004. <http://www.eetimes.com/showArticle.jhtml?articleID=19400052>

- [9] Mastipuram R, Wee EC. (2004). **Soft errors' impact on system reliability** EDN.com & Cypress Semiconductor. Sep 30 2004. <http://www.edn.com/article/CA454636.html>
- [10] Thain D. (2004). **CSE 341 Project 4: Memory Simulator** Operating Systems Principles (class). <http://www.cse.nd.edu/~dthain/courses/cse341/spring2005/projects/memory/>
- [11] Engler DR, Kaashoek F, O'Toole J Jr. (1995). **Exokernel: an operating system architecture for application-level resource management**. Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), Copper Mountain Resort, Colorado, December 1995, pages 251-266. <http://www.pdos.lcs.mit.edu/papers/exokernel-sosp95.ps>
- [12] Furutani K, Arimoto K, Miyamoto H, Kobayashi T, et al. (1989). **A Built in Hamming code ECC circuit for DRAMs** IEEE Journal of Solid-State Circuits, vol 24, no 1. Feb 1989 <http://ieeexplore.ieee.org/iel1/4/584/00016301.pdf>
- [13] Standard Performance Evaluation Corporation. (2001). **SPEC CPU2000 V1.2**. 6585 Merchant Place, Suite 100. Warrenton, VA 20187, USA. <http://www.spec.org/cpu2000/>