Article development led by **acmqueue**
queue.acm.org

**Error-detection and correction features
are only as good as our ability to test them.**

BY STEVE CHESSIN

Injecting Errors for Fun and Profit

“That which isn’t tested is broken.” —Author unknown
“Well, everything breaks, don’t it, Colonel.”
—Monty Python’s Flying Circus

IT IS AN unfortunate fact of life that anything with moving parts eventually wears out and malfunctions, and electronic circuitry is no exception. In this case, of course, the moving parts are electrons. In addition to the wear-out mechanisms of electromigration (the moving electrons gradually push the metal atoms out of position, causing wires to thin, thus increasing their resistance and eventually producing open circuits) and dendritic growth (the voltage difference between adjacent wires causes the displaced metal atoms to migrate toward each other, just as magnets will attract each other, eventually causing shorts), electronic circuits are also vulnerable to background radiation. These fast-moving charged particles knock electrons out of their orbits, leaving ionized trails in their wake.

Until those electrons find their way back home, a conductive path exists where there once was none.

If the path is between the two plates of a capacitor used to store a bit, the capacitor discharges, and the bit can flip from one to zero or from zero to one. Once the capacitor discharges, the displaced electrons return home, and the part appears to have healed itself with no permanent damage, except perhaps to the customer’s data. For this reason, memory is usually protected with some level of redundancy, so flipped bits can be detected and perhaps corrected. Of course, the error-detection and correction circuitry itself must be tested, and that is the main topic of this article.

(If the path is between a current source and ground, then it cannot heal until power is removed. This is called *single event latchup*, which simulates a hard failure, at least until the power is turned off, such as when preparing to remove and replace the apparently failing part. The returned part, of course, will test out as “no trouble found,” frustrating everyone involved. Single event latchup is difficult for software to deal with and will not be discussed further here.)

In addition to the causes of errors mentioned here, transmission lines are subject to noise-induced errors, so transmitted signals are also often protected with redundancy.

As the density of circuits increases, features get smaller; as frequencies increase, voltages get lower. These trends combine to reduce the amount of charge used to represent a bit, increasing the sensitivity of memory to background radiation. For example, the original UltraSPARC-I processor ran at 143MHz and had a 256KB e-cache (external cache). The cache design used simple byte parity to protect the data, which was sufficient as the amount of charge used to hold a bit was large enough that an ionizing particle would drain off only a small amount, not enough to flip a bit.

When this design was scaled up in the UltraSPARC-II processor to run at 400MHz with an 8MB e-cache, however, the amount of charge used to hold a bit

—Monty Python's Flying Circus

was so small that background radiation would easily flip bits, producing on average one flipped bit per processor per year. While that might not seem like a high rate, a customer with 12 systems of 32 processors each would on average experience one failure a day. This is what led to Sun's infamous e-cache parity crisis of 1999 (more on this later; for fun, do a Web search on "e-cache parity").

Since errors, whether transient or permanent, are a fact of life, the system designers in Oracle's Systems organization (what used to be portions of Sun Microsystems) have developed a layered approach to deal with them. At the lowest level is the hardware error-detection circuitry, which records information about the error so that upper-layer software can determine if the error is transient or permanent, or

if the rate of transient errors indicates a failing part. The next layer is error correction, which can be performed by hardware, software, or a combination of the two. The third layer is diagnosis, where the Predictive Self-Healing function of the Solaris operating system determines whether a faulty part is causing the error, and whether that part should be replaced. The final level is error containment, invoked by Predictive Self-Healing when a hard failure can be fenced off so that the system can continue to function with minimal performance degradation, avoiding a disruptive and thus expensive service call.

One always hopes that errors are rare. When they do occur, however, one wants the various layers of detection, correction, diagnosis, and containment to perform flawlessly. Ensuring that requires

testing the various layers, preferably in an end-to-end fashion that imitates the behavior of real errors. Because (as one hopes) errors are rare (if they aren't, you have other problems), waiting around for them to occur naturally is not an efficient testing methodology. Thus, the need for an error injector.

An error injector requires hardware support, because during normal operation hardware only writes good data. (Without hardware support, you can simulate errors by feeding error reports to the upper layers of software, but then you aren't testing the hardware error detectors.) Hardware designers understand this, so they usually provide some means for injecting errors so that they can test their detectors. They don't always understand the environments in which errors will be injected, however.

For example, from the perspective of the hardware designer, testing the detectors during the very controlled environment of power-on self-test (POST) is sufficient, so it isn't a big deal if injecting an error has a side effect of corrupting unrelated data or destroying cache coherency. For the software designer, however, such side effects can render the error-injection hardware useless, or severely restrict the kinds of errors he or she can safely inject.

For example, while the hardware error detector does not care if a cache parity error is detected on a clean or dirty cache line, or by a user instruction or a kernel instruction, the software

by a load instruction or an instruction fetch, so recovery even from an error on a clean cache line was not possible. We were able to recover from parity errors detected by some write-backs, and we definitely improved the kernel's messages when parity errors were encountered. We prototyped confining errors that affected only a user program and not the kernel to just that program (a feature that had to wait for the System Management Facility of Solaris 10 and its process restarter before we could deploy it safely), and we introduced a cache scrubber that used diagnostic accesses to proactively look for parity errors on clean cache lines in a safe

worse job if I were asked to design an ASIC). Instead, we based our error injector on one I had written in 1989 to test the memory parity error-recovery code I had written for Sun's SPARCstation-1. This error injector was modular and table-driven, and easily extensible. Of course, none of the actual low-level error-injection code applied to the UltraSPARC-II, so we hollowed it out and built upon the framework it provided.

The error injector consisted of two parts: a user-level command-line interface (`mtst`), and a device driver (`/dev/memtest`). The command-line interface allowed the user to specify whether the parity error should be injected onto a clean line or a dirty line and whether its detection should be triggered by a kernel load instruction, user-level load instruction, kernel instruction fetch, user-level instruction fetch, write-back to memory, snoop (copy-back) by another processor, or just left in a user-specified location in the cache. (This last was used by another user-level program, affectionately called the alphabomber, to measure the effectiveness of the cache scrubber.)

After parsing and processing its arguments, `mtst` would then open `/dev/memtest` and issue an `ioctl` to it. The parameters passed in the `ioctl` would tell the device driver whether to plant the error in its own space (for kernel-triggered errors) or at an address passed to it by `mtst` (for user-triggered errors) or at a specific cache location (for alphabombing). They would also specify if the device driver itself should trigger the error, and if so by a load instruction, an instruction fetch, a write-back to memory, or a copy-back to a different processor, and whether at trap-level zero or trap-level one. (For obvious reasons, neither `mtst` nor `/dev/memtest` are included in Solaris releases, nor is their source code included in OpenSolaris.)

Assuming the action of the device driver did not deliberately cause a kernel panic, it would return to `mtst`, which, depending upon the parameters with which it was invoked, would either trigger the error (by a load, instruction fetch, write-back, or snoop) or leave it in the cache (for alphabombing).

We later extended the error injector to produce timeouts and bus errors and to inject correctable and uncorrectable memory errors, so we eventu-

layers might. Thus, the error injector must be able to do all combinations.

Injecting E-Cache Errors on the UltraSPARC-II

"Handling errors is just attention to detail. Injecting errors is rocket science."

—me.

While the hardware engineers were working on determining the cause of the e-cache parity errors and then working on a fix, I was asked to lead a project to mitigate with software the effect of the errors. Unfortunately, the UltraSPARC-II used an imprecise trap to report e-cache parity errors detected

fashion (that is, one that would not cause a kernel panic) and flushed them from the cache before they could cause an outage. Whenever the system went idle, we flushed all clean lines, and all error-free dirty lines, from the cache.

Testing all of this required an error injector. While the hardware people had written one, it did not meet our needs; for example, you could only give it a physical address where the error was to be injected and wait for system code to trip over it. In addition, it was neither modular nor easily extensible (after all, it had been written by hardware people; to be fair, of course, I would do an even

ally had complete test coverage of all of the processor error-handling code in Solaris, something that had been lacking prior to this work. (The injection of correctable and uncorrectable memory errors is discussed later.)

The device driver used the diagnostic facilities of the UltraSPARC-II processor to inject the errors into the e-cache. (Similar diagnostic facilities were used by the cache scrubber.) Before I explain how that worked, it will help to understand the following:

- ▶ The UltraSPARC-II uses a 64-byte cache line.

- ▶ A cache line is moved between memory and the e-cache in 8-byte chunks.

- ▶ Each of these chunks is protected in memory by eight bits of ECC (error-correcting code) that can correct any single-bit error and detect any double-bit error (SEC-DED).

- ▶ Each byte of data is protected by a single parity bit when in the e-cache.

- ▶ There are two UDB (UltraSPARC Data Buffer) chips in parallel between the e-cache and main memory, and each UDB converts eight bytes of ECC-protected data at a time to eight bytes of parity-protected data (and vice versa). When a 64-byte cache line is moved from memory into the e-cache or vice versa, each UDB processes four 8-byte chunks.

The interface between the processor and the e-cache is 16 bytes wide. The processor's LSU (load/store unit) contains a control register that includes a 16-bit field called the force mask (FM). Each bit in the FM corresponds to one byte of the 16-byte interface between the CPU and the e-cache. When a bit is zero, a store of the corresponding byte is done with good parity. When a bit is one, a store of the corresponding byte is done with bad parity. The FM bits do not affect the checking of parity on loads from the e-cache.

Injecting a parity error into the e-cache is fairly straightforward. The physical memory address of the desired byte is determined, and the following steps performed:

1. Using its physical address, load the desired byte into a register; this has the side effect of bringing it into the e-cache if it isn't there already.

2. Disable interrupts.

3. Set LSU.FM to all ones.

As the density of circuits increases, features get smaller; as frequencies increase, voltages get lower. These trends combine to reduce the amount of charge used to represent a bit, increasing the sensitivity of memory to background radiation.

4. Store the desired byte back to its physical address. (If for some reason the containing cache line got displaced from the cache after the load, then this will bring it back into the cache.) The targeted byte will be written back into the cache line with bad parity.

5. Reset LSU.FM to zero.

6. Reenable interrupts.

Now that the desired byte is in the e-cache with bad parity, the latent error can be triggered via several mechanisms: data load in user or kernel mode, instruction fetch in user or kernel mode, displacement flush to cause a write-back, access from another CPU to cause a copy-back, and so on.

Interrupts must be disabled for the duration that the LSU.FM is not zero; otherwise, if an interrupt occurs and the interrupt handler (or any code it invokes) performs a store, then undesired parity errors will be introduced into the cache and triggered unpredictably.

This six-step sequence is used to inject e-cache parity errors at locations corresponding to specific physical memory addresses, kernel virtual addresses, or user virtual addresses. (Virtual addresses are translated to their corresponding physical addresses by the memtest device driver.) To simulate bit flips caused by background radiation, however, we would like to inject an e-cache parity error at an arbitrary e-cache offset, without regard to the physical memory address corresponding to the e-cache line.

Fortunately, the LSU.FM field also applies to stores to the e-cache using diagnostic accesses. Unfortunately, diagnostic loads and stores work only with 8-byte quantities, not with single bytes. In order to affect just a single byte, we must set only the one bit in LSU.FM that corresponds to the byte we want to change. The sequence in this case then becomes:

1. Disable interrupts.

2. Fool the instruction prefetcher (see below).

3. Set the desired bit in LSU.FM to one.

4. Load the containing eight bytes into a register with a diagnostic load.

5. Store the containing eight bytes back into the e-cache with a diagnostic store.

6. Reset LSU.FM to zero.

7. Reenable interrupts.

The only tricky part is preventing the contents of the e-cache from changing out from under us between the load and the store. The worst that snoop activity can do is change the state of a line from exclusive to shared, or from valid to invalid. As snooping cannot change the data itself, just the state in the tag, no harm is done if a snoop occurs between the load and the store.

However, there is one thing that can change the data in the cache between the load and the store. The processor contains an instruction prefetcher—one that is always on and whose behavior is not well documented in the *UltraSPARC I & II Users Manual*. The prefetcher is constantly moving instructions from the processor's i-cache (instruction cache) into the processor's instruction buffer. If the address of the next instruction to be prefetched misses in the i-cache, instructions will be brought in from the e-cache; if the address also misses in the e-cache, then the containing cache line will be brought into the e-cache from memory, displacing what was already there. If this e-cache fill happens to replace the line containing the byte we want to corrupt, and if the fill happens between the diagnostic load and the diagnostic store, we will write eight bytes of stale data into the e-cache (along with bad parity on one of them); this could cause an unexpected failure later if the line is reexecuted as an instruction. (Although we expect the byte with bad parity to cause an eventual failure, we want the failure to be the one we intended, not one we didn't intend.)

To prevent this, the prefetcher must be fooled into not prefetching for a while. Though this is possible—and in fact fairly easy to do—the procedure is not documented. The technique to use had to be obtained from the processor pipeline expert. In fact, if he hadn't informed us of this exposure, we would have had a hard-to-debug problem with the injector.

To fool the prefetcher, we statically position at the beginning of a cache line the code sequence that sets LSU.FM, issues the load and store, resets LSU.FM, reenables interrupts, and returns to the caller. When this routine is called, it disables interrupts and then branches just beyond the above sequence to a series of no-ops, enough

to fill the instruction buffer. The last instruction in this sequence branches back to the instruction that sets LSU.FM. Thus, when we get to the load of the load/store pair, the cache line that contains these instructions is already in the e-cache and has either already displaced the original target (so we will be injecting an error on top of our e-cache-resident code) or is in a different cache line than our target. In either case, the instruction prefetcher “sees” that the instructions (including the no-ops) that follow the load/store pair are already in the instruction buffer, so it temporarily has nothing to do. This prevents any lines from changing in the middle of the execution of the load/store pair. (This is the “rocket science” part of error injection.)

Of course, what would have really been nice would have been a control to turn off the instruction prefetcher.

Injecting Memory Errors on the UltraSPARC-II

“The horror of that moment,’ the King went on, ‘I shall never, NEVER forget!’ ‘You will, though,’ the Queen said, ‘if you don’t make a memorandum of it.’”

—Lewis Carroll,

Through the Looking Glass

Injecting memory errors on UltraSPARC-II systems is more difficult than injecting e-cache errors. As previously described, while the e-cache uses byte parity, memory uses eight bits of ECC to protect eight bytes. Data always moves between memory and the CPU subsystem (processor, two UDB chips, and e-cache) in 64-byte blocks, transferred in four 16-byte chunks. Each UDB handles eight bytes at a time, converting eight bytes with good ECC into eight bytes with good parity and vice versa.

Each UDB has a control register that contains an 8-bit FCBV (force check bit vector) field and an F_MODE (force mode) bit. When the F_MODE bit is set, the UDB uses the contents of the FCBV field for the ECC value on all outgoing (to memory) data, instead of calculating good ECC.

Since the FCBV field (when used) applies to all data going through the UDB, and since the smallest granule of transfer is 64 bytes, it is impossible to force bad ECC on just one arbitrary 8-byte extended word. (This means we cannot alphabomb CEs into arbitrary

locations.) Generating a single CE (correctable error) or UE (uncorrectable error) requires that the four 8-byte extended words passing through a given UDB start off as identical, so that they all share the same good ECC value.

Generating a CE or UE is typically done as follows:

1. Quiesce snoop activity, as snooped data goes through the UDBs.
2. Disable interrupts.
3. Set FCBV in the UDBs with the common good ECC value, and set their F_MODEs.
4. Load the desired 8-byte chunk into a register; this has the side effect of bringing it into the e-cache if it isn't there already.
5. Flip one (CE) or two (UE) bits in the register.
6. Store the now-modified 8-byte chunk; it will store into the cache and put the cache line into the modified state.
7. Displacement flush the cache line back to memory. The UDBs will convert each eight bytes with parity into eight bytes with ECC, but for the ECC bits they will use the value in the FCBV, which will be good for all but the modified chunk.
8. Clear F_MODE.
9. Enable interrupts.
10. Allow snoop activity.

(Although we could have confined the setting of FCBV and F_MODE to just the UDB handling the targeted location, it was easier to program them both identically.)

Snoop activity has to be quiesced; otherwise, any CPU or I/O device obtaining data out of this CPU's e-cache while the UDB's F_MODE bit is set will get bad ECC. Since I/O is difficult to quiesce, this is done by “pausing” all the other CPUs (by telling them to spin in a tight loop), and then flushing the cache so that the only owned line will be the one that we modify.

To inject a single CE at an arbitrary location, the UDB design should have included a “trigger” or “mask” field to indicate on which extended word(s) the FCBV field would be applied. This field could be, for example, an 8-bit mask, with one bit for each 8-byte chunk. (One UDB would use the even bits and the other would use the odd bits; this arrangement would make programming simpler.) The UDB would have to count the chunks going through it when the

F_MODE bit was set and apply FCBV to only those extended words that had the corresponding “trigger” bit(s) set.

Alternatively, the design could have included eight sets of FCBV fields (four in each UDB), each with its own F_MODE bit, so that arbitrary mixes of CEs, UEs, and good data could be planted at any location.

Other Uses of Diagnostic Access

“I’m running a Level 1 diagnostic.”

—Lt. Commander Geordi La Forge, in
Star Trek: The Next Generation

As illustrated earlier, diagnostic access to the e-cache and the memory interface chips is extremely important to

in the hardware. For example, while diagnostic access to the e-cache does not return the parity bits, the parity check logic works and sets the PSYND (parity syndrome) bits in the AFSR (Asynchronous Fault Status Register) as appropriate. (The 16 PSYND bits correspond to the 16 bytes in the interface between the processor and the e-cache. If a byte contains a parity error, the corresponding PSYND bit is set to one.) Thus, diagnostic access to the cache allows the parity bits to be inferred, if not observed directly.

It is important to note that the use of diagnostic access by the error injector and the cache scrubber depends

occurs the system board can send it to just those processors that contain copies of the snooped cache line, and not interfere with the performance of the processors that do not contain copies. Diagnostic access to DTAGs interferes with the maintenance of cache coherency, such that if an invalidate request came in at the same time as the diagnostic access, the invalidate request would be lost. (The request need not be for the particular line; all coherency traffic is ignored while the diagnostic request is being processed.)

This behavior makes it impossible to write a software DTAG scrubber, as the scrubber cannot determine if a line contains a latent error without risking the loss of system coherency.

Note that deciding whether to preserve coherency on a diagnostic access is an example of one of the many decisions a chip designer must make. Prior to Sun’s e-cache parity crisis, these decisions were made by the hardware designers without consulting the software error-handling experts. Since that crisis, error and diagnostic reviews of new chips are a required part of the hardware design cycle.

These reviews are joint meetings of the chip designers and the software people responsible for error handling, diagnosis, and containment. They are held early enough in the design process so that any deficiencies in the treatment of errors by the hardware (such as a failure to capture important information) can be corrected, and suggestions of improvements can be incorporated.

Other Methods of Error Injection

“Doctor, it hurts when I do this.”

“So don’t do that.”

—Henny Youngman

Hardware engineers have developed other methods for injecting errors, some more usable than others. For example, having learned from our e-cache parity experience, subsequent processors in the UltraSPARC line, beginning in about 2001 with the UltraSPARC-III, protect the e-cache with true ECC. In the UltraSPARC-III 16 bytes of data are protected by nine bits of ECC, and this same scheme is used to protect data in memory as well. (ECC is checked as data is moved from the e-cache to memory; single-bit errors are corrected and double-bit

error injection. Without the ability to use diagnostic loads and stores during normal system operation, injection of errors would be impossible.

Diagnostic access is also used in error prevention and correction, as the cache scrubber uses diagnostic loads to determine if a latent error is present, and to determine when lines should be displaced from the cache.

Diagnostic access is also used after a failure occurs, to read the contents of the affected cache line to aid in offline diagnosis. For this reason, it is important that diagnostic access provide visibility to all the bits, as they are stored

on their not interfering with normal system operation. In particular, system coherency must be maintained while the diagnostic operation is in progress.

The caches of UltraSPARC-II obey this requirement. Diagnostic access to the cache by the CPU does not interfere with the cache’s response to coherency traffic. Snooping continues, and requests to invalidate cache lines are processed normally.

Contrast this with the Sun Enterprise 10000 system board DTAG (dual tag), which contains a copy of the tag information in the four processors on the system board. Thus, when a snoop

errors are rewritten with a special syndrome. Similarly, ECC is checked as data is moved from memory to the e-cache; single-bit errors are corrected, but double-bit errors are written into the e-cache as is.)

Injecting memory errors in the UltraSPARC-III is similar to that in the UltraSPARC-II; a control register contains an FM bit and a forced ECC field. When set that ECC value is used instead of calculated ECC when data moves from the e-cache to memory.

For injecting errors into the UltraSPARC-III e-cache, the hardware engineers tried to do something similar; another control register contains an

registers: four to hold 32 bytes of data (a half-cache line consisting of two 16-byte ECC-protected chunks) and a fifth register to hold the two 9-bit ECC fields protecting the respective chunks. One set of diagnostic loads and stores moves data between the e-cache and the staging registers 32 data bytes and 18 ECC bits at a time; another set moves data between a given staging register and an integer register. This allows the error injector to flip any combination of data and ECC bits.

Conclusion

"Software. Hardware. Complete."

Since the e-cache parity crisis, error

advantage of error injection technology to improve the handling, diagnosis, and containment of errors by their respective systems, as having the hardware and software people all in a single organization allows the necessary continuous interaction between them as new hardware and software is developed. When hardware and software development is divided among different organizations, as it is in the Windows, VMware, and Linux worlds (or, alternatively, the Intel and AMD worlds), exploiting error injection technology for product improvement is much more difficult.

Acknowledgments

Much of this article is based on the work of the circa-1999 Solaris Software Recovery Project. That project was the result of cooperation between and hard work by many individuals from across Sun, including Mike Shapiro, Huay Yong Wang, Robert Berube, Jeff Bonwick, Michael Christensen, Mike Corcoran, John Falkenthal, Girish Goyal, Carl Gutekunst, Rajesh Harekal, Michael Hsieh, Tariq Magdon Ismail, Steven Lau, Patricia Levinson, Gavin Maltby, Tim Marsland, Richard McDougall, Allan McKillop, Jerriann Meyer, Scott Michael, Subhan Mohammed, Kevin Normoyle, Asa Romberger, Ashley Saulsbury, and Tarik Soydan. Robert Berube in particular did much of the initial coding of the UltraSPARC-II error injector.

I also want to thank Mike Shapiro and Jim Maurer for reviewing early drafts. Their suggestions have improved this article. Any errors that remain are solely my responsibility.

FM bit and a forced ECC field, only the forced ECC in this register is used whenever data is written into the e-cache. This would have been difficult to use, as stores do not write data directly into the e-cache, but into a w-cache (write cache). The data in the w-cache is not merged with that in the e-cache until the line is displaced out of the w-cache, and that is difficult to control. Fortunately, we did not have to use this mechanism, as the hardware engineers provided something even better: direct access to the raw bits in the e-cache, both data and ECC.

This mechanism uses five staging

injection has become a core competency of what is now Oracle's Systems organization. As new processors and their supporting ASICs are designed, error and diagnostic reviews make sure they have the appropriate ability to inject errors into their internal structures, and the error injector is enhanced to inject those errors so that we can test our error-handling, diagnosis, and containment software in an end-to-end fashion.

Of course, companies such as IBM and Oracle that control both the hardware they sell and the software that supports it are best positioned to take

Related articles on queue.acm.org

Self-Healing in Modern Operating Systems

Michael W. Shapiro

<http://queue.acm.org/detail.cfm?id=1039537>

A Conversation with Jeff Bonwick and Bill Moore

<http://queue.acm.org/detail.cfm?id=1317400>

You Don't Know Jack about Disks

Dave Anderson

<http://queue.acm.org/detail.cfm?id=864058>

Steve Chessin (steve.chessin@oracle.com) is a senior principal software engineer in the Systems Group Quality organization of Oracle Corporation, Menlo Park, CA.

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.