A Deep Technical Review of nZDC Fault Tolerance

Minli Liao
University of Cambridge
Cambridge, United Kingdom
ml2076@cam.ac.uk

Sam Ainsworth
University of Edinburgh
Edinburgh, United Kingdom
sam.ainsworth@ed.ac.uk

Lev Mukhanov
Queen Mary University of London
London, United Kingdom
l.mukhanov@qmul.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, United Kingdom
timothy.jones@cl.cam.ac.uk

Abstract

Faults within CPU circuits, which generate incorrect results and thus silent data corruption, have become endemic at scale. The only generic techniques to detect one-time or intermittent soft errors, such as particle strikes or voltage spikes, require redundant execution, where copies of each instruction in a program are executed twice and compared.

The only software solution for this task that is open source and available for use today is nZDC, which aims to achieve "near-zero silent data corruption" through control- and data-flow redundancy. However, when we tried to apply this to large-scale workloads, we found it suffered a wide set of false positives, negatives, compiler bugs and run-time crashes, which meant it was impossible to benchmark against. This document details the wide set of fixes and workarounds we had to put in place to make nZDC work across full suites. We provide many new insights as to the edge cases that make such instruction duplication tricky under complex ISAs such as AArch64 and their similarly complex ABIs. Evaluation across SPECint 2006 and PARSEC with our extensions takes us from no workloads executing to all bar four, with 2× and 1.6× geomean overhead respectively relative to execution with no fault tolerance.

CCS Concepts: • **Software and its engineering → Software reliability**.

Keywords: Software reliability, Compilation

ACM Reference Format:
Minli Liao, Sam Ainsworth, Lev Mukhanov, and Timothy M. Jones. 2025. A Deep Technical Review of nZDC Fault Tolerance. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3708493.3712688

CC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1407-8/25/03
https://doi.org/10.1145/3708493.3712688

1 Introduction

The threat of silicon faults, causing processors to produce incorrect results, has rapidly become a real problem at scale, with data-centre operators observing a litany of real-world silent data corruptions [10, 15] in spite of today's existing Reliability, Availability and Serviceability (RAS) mechanisms[1, 13, 21, 29]. Full lockstep core duplication [2, 16] has only seen wide deployment in highly safety critical environments such as automotive due to its expense, so software mechanisms for repetition and verification of execution are desirable and have seen much research effort [5–7, 9, 17, 26–28, 30, 32, 33].

Despite this wide set of work, the only tool we are aware of that is available with a source-code release [19] is nZDC [6]. nZDC runs duplicate instruction copies on duplicate shadow registers inside each thread, along with careful handling of memory and control flow to achieve "near-zero silent data corruption", in order to detect transient execution faults, where units intermittently produce the wrong value. It is implemented as an LLVM backend pass for the AArch64 ISA. However, the original paper evaluates its correctness only on simple Mibench [12] workloads under gem5 simulation.

Here we look at nZDC's behaviour on real systems and diverse workloads. We discover that even at the scale of complexity of benchmarks such as SPEC CPU2006 [14] and PARSEC [3], the only publicly available CPU soft-error redundancy mechanism hits a wide set of both compilation failures, memory exceptions, false positives and false negatives, that greatly harm its viability.

This paper analyses the problems we encountered when applying the nZDC implementation to complex benchmarks. We carefully categorise these and explain why the errors we observed lead to interesting new insights where redundant execution is tricky, prone to edge cases, or is truly intractable. We give the steps we took to fix them where feasible, and workarounds where not. We hope that this paper provides some insights on what to be careful of to researchers developing similar techniques, and provides some hints on how to debug to researchers attempting to reproduce and extend similar techniques from the literature. We also release our large set of modifications that take nZDC from executing no workloads from SPEC and PARSEC (and only one compiling)

to executing all bar four, giving analysis of the remaining failures and false negatives where full behavioural fixes were infeasible, impossible, or require novel research effort and/or complex re-engineering.

We introduce the nZDC scheme in section 2. We then present the high-level problems we have discovered, their root causes, fixes, workarounds and lessons learned when trying to use it in section 3. Performance results and success rates in execution from our evaluated workloads are given in (section 4), followed by a discussion of other works in the area (section 5). The source code for our modified nZDC is available at https://github.com/CompArchCam/CC25-AE, and archived at https://doi.org/10.5281/zenodo.14678385.

2 Overview of nZDC

nZDC is a software-based reliability technique that detects faults through instruction duplication. Some registers are reserved and used as redundant copies (shadows) of other registers, and instructions are duplicated with shadow registers used in the copy. The values from the original and the shadow registers are then compared, where any differences would indicate an error occurred. Specifically, with nZDC, all computation, memory-read, compare, and direct-branch instructions are duplicated, and stores and conditional branches are protected with custom strategies.

To set up shadow register values, values from original registers are copied to the corresponding shadow registers at the beginning of the main function and after each function call, as shown in figure 1[1]. Instructions except stores and conditional branches are duplicated using shadow registers. The checking by comparing the original register value with that from its shadow version happens before function calls (and also immediately after stores as discussed below). A conditional branch is inserted after each check, to branch to error handling code if an error is detected.

Stores are protected by a following *checking load* that uses the shadow register for the memory-access base address, but not for the destination (as shown in figure 2). Then, the re-loaded value is compared against the value in the shadow register to check if the store was successful. A conditional branch is inserted after the compare to take the execution to the error handling code in the case of error where the re-loaded value differs. The base address register's value is also compared against that in its shadow register to check if the store was to the correct memory location.

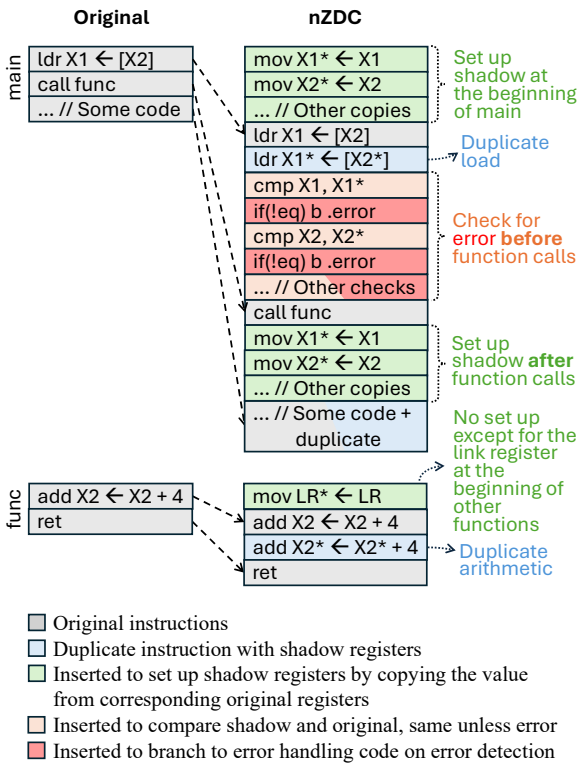

Figure 1. Shadow registers get their values from original registers at the beginning of `main` and after each function call, duplicate instructions using shadow registers are inserted for redundant execution, and register values are checked before each function call (X1*, X2* and LR* are shadow registers of X1, X2 and LR respectively).

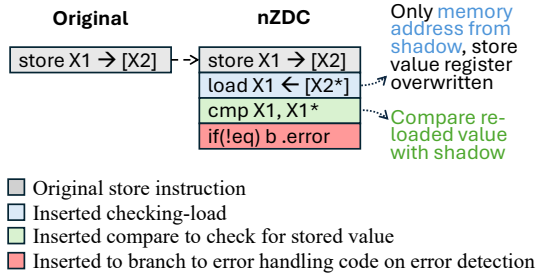

Figure 2. nZDC protects stores through inserting checking-loads (X1* & X2* are shadow regs of X1 & X2 respectively).

For conditional branches, two extra registers are reserved for checking[2]. As shown in figure 3 (adapted from fig. 6 in

[1]Note that the copying after calls means that the return values are unchecked. This is the behaviour seen from the implementation in the repository [19], and prior work SWIFT [26] mentions that while this is a vulnerability, shadowing the return register would require modifying the calling convention to accommodate multiple return registers, hence values are copied after return to avoid calling-convention modifications.

[2]These registers are correctly reserved (X27 and X28 in the source code, currently) in the original implementation, and here. However, due to calling-convention issues we found working with LLVM in practice (sections 3.3.4 and 3.4.3), these registers occasionally get overwritten around call boundaries even though no user code stores in them, and there are too few registers that never get overwritten to reserve two other, safe registers for this purpose. Since branch checking is not actually implemented (section 3.2), this

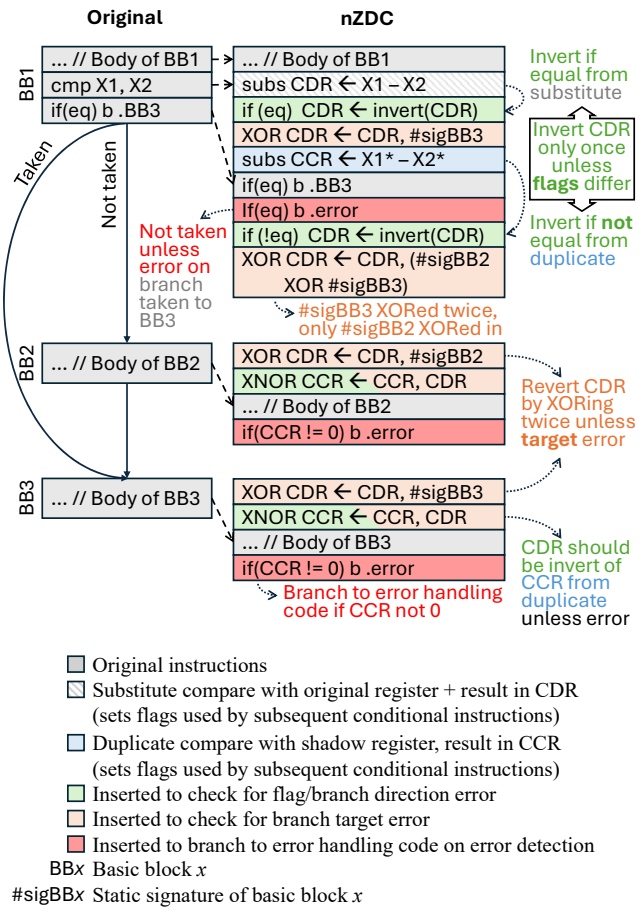

Figure 3. Example of the nZDC transformation to protect conditional branches with reserved `CDR` and `CCR` registers (`X1*` and `X2*` are shadow registers of `X1` and `X2` respectively).

Didehban et al. [9]), the destination of the condition-flag-setting instruction is changed to be one of the extra reserved registers (referred to as the `CDR`), and then conditionally inverted to check for errors in the flags. The other extra reserved register is used in the duplicate condition-flag-setting instruction that uses the shadow registers for its source (`CCR`). Additional instructions are also added to check for branch direction and target error by adding in a static signature.

3 Problems of nZDC

While trying to use the public source code for nZDC [19] and apply it to SPEC and PARSEC benchmarks[3], we identified several problems, both with the source code itself and with the fundamental mechanism. We first detail the key concepts and insights behind each issue before giving examples, root-cause analysis and either workarounds that allow execution without full error detection, or fixes when easily feasible.

We start with problems with the design of nZDC. First, on Arm ISAs, there are issues around word length between types of instructions that cause a store-and-reload of supposedly the same data to generate false positives on being checked (section 3.1). Second, flags for conditional branches can be set in multiple ways in Arm, and this causes interference between control-flow and data-flow checking (section 3.2). Third, oversights around complexities in the application-binary interface (ABI), such as calling-convention behaviour when moving to library code not protected by nZDC, or on exceptions, causes a variety of unintended consequences (section 3.3). We then look at a set of interesting corner cases that break inside the implementation itself (section 3.4), before briefly listing other similar issues with less fundamental insights (section 3.5). Finally, we also encountered a couple of problems that are not due to nZDC (e.g. compatibility of new tools with the old code base), and discuss solutions in detail for these as well (section 3.6).

3.1 Checking-Reload False Positive

This section describes a fundamental problem that arises from the design of nZDC under its target AArch64 ISA.

In the method for store checking, nZDC assumes that the checking-load is a perfect symmetry of the checked store to load back what is stored, hence the register with the reloaded value should match that of the shadow of the stored value. However, as shown in figure 4, this assumption is not always correct, because of subtleties around result length in ARMv8 AArch64 (nZDC's [6] target ISA). Stores of sub-register size (e.g. STRB or STRH) will only store a portion of the value in the (32- or 64-bit) register without changing the register value. However, with AArch64, loads of sub-register size will have the loaded value extended to fill up the full register size. 32-bit loads also fill the top 32 bits of 64-bit registers with 0s. This means that even if the store is correct and the stored bits are the same as before, the checking load will

is innocuous, though we expect challenges were a true branch-checking implementation attempted.

[3]It can be argued that all research artefacts are only tested on the workloads used in the original authors' experiments, and so one should expect minor issues when applying techniques to new code. The reason we believe that applying nZDC to SPEC CPU2006 and PARSEC is instructive, and why this process produced enough interesting bugs to fill a full paper, is in major part due to the simplicity of the MiBench [12] suite originally evaluated [6].

While MiBench is easy to compile, using a small subset of only the C language, making it suited to embedded systems research, and is easy to simulate in a short amount of time in e.g. gem5 as the original nZDC authors used, we saw significantly higher code diversity and much longer run-times from SPEC and PARSEC. For example, the `qsort` workload in MiBench is really just a call to the C library function of the same name. This is one reason why SPEC and PARSEC are more widely used in the literature, and the ability to run widely accepted workloads is critical to allow future research to perform acceptable comparisons and evaluations. This does not mean our version has complete coverage for all types of code (as we discuss, we do not even cover all SPEC workloads, with some failing for documented reasons), but SPEC and PARSEC together are diverse enough for us to discover what we hope are the bulk of interesting edge cases.

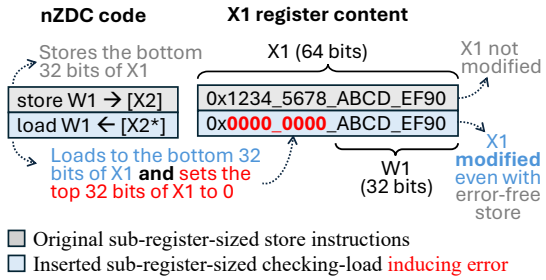

Figure 4. Sub-register-sized checking-load induces error because it also modifies the bits that were not loaded in the destination register (X2* is the shadow registers X2).

still modify the other bits of the register value, resulting in differences with the copy in the shadow register.

We provide an example and workaround here. Comments are shown as blue-coloured text above lines of code.

Cause nZDC re-loads stored values to the same register to check for errors, however, sub-register size re-loads may change upper register bits beyond the loaded size.

Example From perlbench, in the generated assembly file (assume x24 shadows x1, x12 shadows x19).

```
// Original store, x1 has non-zero upper 32-bits
str w1, [x19, #48]
// Inserted checking-load, upper 32-bit of x1 now
// changed to zero
ldr w1, [x12, #48]
// Check for store success, check only lower bits
sub w25, w1, w24
// True negative store success
cbnz w25, .LBB1162_154
// Duplicate shift right
lsr x24, x24, #32
// Original shift right with zeroed out upper bits,
// lower bits after shift doesn't match x24
lsr x1, x1, #32
// Original store with wrong value
str w1, [x19, #52]
// Inserted checking-load
ldr w1, [x12, #52]
// Check for store success
sub w25, w1, w24
// False +ve due to checking-load incurred error
cbnz w25, .LBB1162_154
```

Error Manifest as segmentation fault or as false-positive error detected during execution.

Reason Not fixable without reserving an extra register because the AArch64 ISA defines that a 32-bit load would write 0s to the upper 32 bits of a 64-bit register, and 16/8-bit loads would sign/zero-extend. There is no instruction in the ISA that can load sub-register size values without overwriting the other bits of the destination register. The original register value cannot be recovered unless an additional register is used to cache that information, which would result in a store checking method different in theory to that of nZDC.

	Original nZDC	Implemented workaround	Potential fix with XOR
Original store	store W1 --> [X2]	store W1 --> [X2]	store W1 --> [X2]
Checking load	load W1 <-- [X2*]	load W1 <-- [X2*]	load W_new <-- [X2*] *(W_new/X_new denotes the extra reserved register)*
Extra instruction for checking	N/A	mov X1 <-- X1*	xor X_new <-- X_new, X1* *(lower bits now 0 if no error)*
Check for store success	cmp W1, W1*	cmp W1, W1*	tst X_new, mask *(alias of "ands XZR, X_new, mask", sets flag Z if result is 0; mask has 0s for higher bits, 1s for lower bits))*
Other code	...	...	...
Check before call	cmp X1, X1* (false +ve)	cmp X1, X1*	cmp X1, X1*

Figure 5. The original nZDC store check (which fails on checking upper bits), our implemented workaround (which has the same overheads as a proper check, but overwrites the incomplete shadow to correct the upper bits, so does not find errors), and a full fix, currently unimplemented.

Workaround Workaround by inserting instruction to copy register value from shadow back to original register to suppress false-positive errors in LLVM.

Residual False Negative The above workaround preserves overheads but leaves stores unchecked, meaning the wrong value could propagate to memory.

Lesson Loads and stores are not perfect symmetries, and loads will change the register value beyond the loaded bytes. When loading back the stored bytes and checking for correctness, the other bytes of shadow register need to be preserved.

Sub-word loads and stores are extremely common, and so the majority of SPEC and PARSEC workloads were affected by false positives as a result. On top of false positives, the checking-load is actually inducing errors: the perlbench benchmark for example has errors in the output with checking-loads added, even with all error detections ignored, because the original register value used by the execution that is independent to checking has been changed.

A fix for this behaviour requires being able to hold and restore the upper bits that the load should not have overwritten (figure 5). This could be done by replacing the original compare instruction (e.g. the `cmp` instructions in the example[4] with an `xor` of the reloaded value in a new register with the stored value in the original register (which should be

[4]Though the original paper [6] describes this check as being a `cmp` instruction followed by a `b.ne`, as in figure 5, the actual source code they provide [19] implements this instead as `sub` followed by `cbnz`, as in the error example we give. The reason this matters is that the former only affects flags, whereas the latter requires the reservation of an extra register: X25/W25. We are unsure why this difference exists, though this would allow us to use the same X25 register instead for the reload and xor in our potential fix (X_new in figure 5). Still, in general NZDC is very register-limited

zero in the lower bits if correct), followed by an ands with mask that also sets the condition flag, to verify the lower bits are all zero. Since this has the same instruction overhead as our implemented workaround, gives trouble with register reservation (section 3.3.4), and requires very different code sequences for different integer sizes and thus a significant engineering rearchitecting of the codebase, we do not provide the full fix in our source release.

3.2 Branch-Transformation Inaccuracy

This section describes the problems that arise from the description of the nZDC transformation for conditional branches. Since there is no implementation in the first place, we simply describe our effort to work around this problem in the absence of this implementation to obtain workable binaries.

The nZDC transformation for conditional branches (described in the paper [6] but not implemented in the open-source version [19][5]) assumes that conditional branches use `cmp` instructions to set up the condition flags. `cmp` is an alias of `subs` (subtraction that also sets the conditon flag) with `XZR` as the destination register. And since `XZR` is always 0, this essentially means that the flags are set but the result of the subtraction is discarded. This is why replacing the destination register with a reserved register for error checking as described in the paper is feasible: the discarded result is being re-used as part of error checking. However, having `XZR` as the destination register is just a special case of the `subs` instructions (albeit a very common case): flags for conditional branches can be set with `subs` or other flag-setting instructions where the result is not discarded but recorded in a general purpose destination register as well; we saw this in all but two of the tested SPEC and PARSEC benchmarks. Simply replacing the destination register in these cases leads to errors since the original destination never gets the result. As shown in figure 6 where the destination register is `X1` and not `XZR`, substituting the destination with `CDR` without copying the value back to `X1` results in `X1` value error. A more general transformation would need to copy the result from the reserved register to the original destination register before any instruction that modifies its value, to ensure correctness (and same for the shadow of the destination and the duplicate instruction using it). Here we give an example

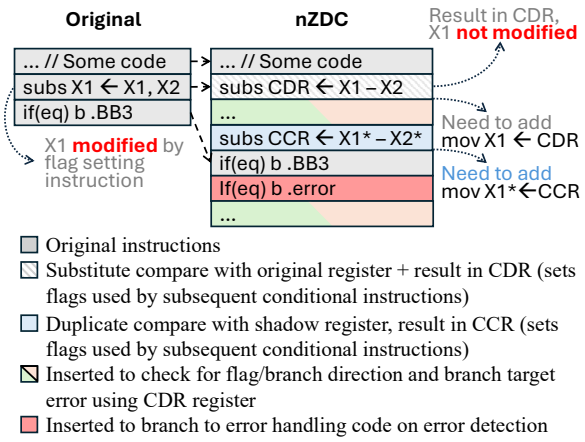

Figure 6. nZDC's conditional-branch check fails when the `CDR`/`CCR` registers are not substituting XZR.

and show how we work around this problem in the absence of the transformation for conditional-branch checking being implemented.

Cause nZDC duplicates arithmetic instructions with the redundant registers to check for errors. However, some arithmetic instructions that also set flags are not duplicated, resulting in a value difference between the original and the redundant registers (this is related to the implementation of the transformation to protect conditional branches).

Example From bzip2, in the generated assembly file:

```
// w0 modified but no duplication
subs w0, w3, #1
```

Error Manifests as a false-positive error detected during execution. The specific error depends on the content of the error recovery block (e.g. segfault if the error-recovery block accesses an invalid memory address).

Workaround Workaround through adding duplication of a non-flag setting version of the same arithmetic instruction in assembly.

Residual False Negative Branches remain unchecked due to a lack of implementation on control-flow checking in the original repository. Also, flags in general are not checked for correctness.

Lesson Very common special cases should not be considered the general case, as even edge cases will be common in large workloads.

3.3 Library and Calling-Convention Compatibility Issues

This section describes problems that arise from the design of nZDC and we can see a clear solution or are concerns that we have about applying nZDC more generally to other

(section 3.3.4): we expect that combinations of full branch checking (requiring two registers, `CCR` and `CDR`, that are never clobbered by saving and restore) and load checking (requiring reserving one further safe register) will be infeasible without changing the calling convention or at the very least resolving issues with spurious saving of registers in the LLVM backend (section 3.4.3).

[5]A somewhat similar conditional-branch mechanism is, however, implemented in another of the original authors' repositories [18]. The control-flow error-detection scheme implemented there is NEMESIS [8], which is different from nZDC and is implemented on the OR1K target. However, NEMESIS does insert duplicate branches that uses the shadow registers and and branches of opposite conditions, which may provide some insight on how to implement nZDC's branch control flow.

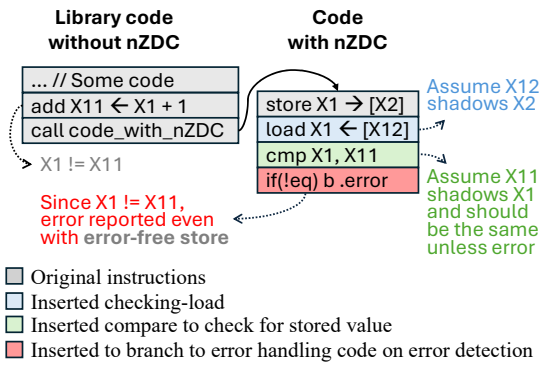

Figure 7. nZDC checks report false positives when called from library code compiled without nZDC transformation.

ISAs while AArch64 may not specifically have a problem. We again describe our effort to work around or solve these problems to obtain workable binaries.

These are typically due to linking with precompiled libraries. In theory, nZDC could be redesigned to only work if all code is precompiled with nZDC, including linked code. This would give more freedom in the use of shadow registers, avoiding several issues described below, as well as giving higher code coverage by actually covering all code. However, we believe that compatibility with linked code is important, and the original authors agreed [6]: all of their experiments evaluated with precompiled libraries not covered by nZDC, and the authors claimed such a setup should not break program behaviour. Forcing the recompilation of libraries makes the use of this tool much more complex, not matching typical compilation workflows.

3.3.1 Library Code Destroys Shadows. Reserving registers for shadowing requires compile-time transformations, which means that any linked library code that was not compiled with the transformation does not have the same agreement on which registers are shadows. In addition to the lack of error detection coverage for such library code, the disagreement on what each register is used for leads to broken code (this is similar to a calling convention). As shown in figure 7 for example, when code without the transformation calls user code with the transformation, the shadow registers are not set up to have the same value as the corresponding original register, which leads to differences in comparison and results in false positives in error checking.

Cause The linked library code did not go through the nZDC transformation, hence what are considered shadow registers in nZDC do not hold a copy of the value of those in the original registers.

Example From blackscholes's C++ source, the `bs_thread` function pointer is passed to the `pthread_create` function. When compiling, `bs_thread` goes through nZDC transformation from the custom compiler, `pthread_create` is linked in from an existing library that was not compiled with nZDC transformation.
In the generated assembly file (assume `bs_thread` called from `pthread_create`, x21 shadow sp, x12 shadow x19):

```
// start of bs_thread
bs_thread:
  ...
  // Original store
  stp x20, x19, [sp, #-32]!
  // Inserted load
  ldp x20, x19, [x21, #-32]!
  // Check for store success, but pthread_create
  // did not keep the values of x19 and x12 the
  // same
  sub x25, x19, x12
  // False positive
  cbnz x25, .LBB2_6
```

Error Manifests as a false-positive error detected during execution when library code calls nZDC instrumented user code because redundant registers hold values different from the original registers.

Workaround Workaround by either removing the nZDC insertion in code called by library code (for anything used before `main`), or re-copying the values from original registers to redundant registers at the beginning of the function (for anything only used after `main`) in LLVM[6].

Residual False Negative Code in an nZDC compilation unit that is called from outside the unit is not covered by nZDC—though given nZDC does not cover library code anyway, this is not relatively speaking a huge difference in coverage.

Proper Fix A proper fix would require re-compiling all library code with the nZDC transformation as well, unlike in the method used by the original paper [6].

Lesson Properties that must be preserved across function calls need to be respected by both the caller and the callee. And library code can be the caller as well, not only the callee. Techniques may allow library code to be unprotected, but care must be taken so that it does not break the error detection in the protected user code.

3.3.2 Library Calling-Convention Violation. The choice of which registers to use as shadows, or reserved for branch or load checks, may also break the calling convention. In AArch64, some of nZDC's reserved registers are inevitably callee-saved registers, since there are too few remaining registers to form a full shadow otherwise. This means that changing their values directly without saving and restoring

[6]This is implemented by altering the nZDC pass to look for known-bad function names from SPEC and PARSEC. For new workloads, similar lists would need to be created by profiling when false positives occur.

breaks the calling convention (e.g. in nZDC, as shown in figure 8). While there is no problem with code that follows the agreement that these registers are always used as shadows, this becomes a problem when working with library code that does not follow this agreement: when a library code without the transformation calls the user code with the transformation, it expects the callee-saved registers to retain their values, but the callee would have changed their values for shadowing, assuming they are reserved and never used for other purposes, resulting in errors in the program[7].

Cause Linked-library code did not go through the nZDC transformation and assumes that callee-saved registers keep their values as per the calling convention, but nZDC uses those registers as shadows and modifies their values without saving and restoring.

Example From h264ref, in the C++ code, the function pointer of `compare_pic_by_pic_num_desc` is passed to the `qsort` function. When the former function goes through the nZDC transformation from the custom compiler, `qsort` is linked in from an existing library that was not compiled with the nZDC transformation.
In the generated assembly file (assume x23 shadows x0, x24 shadows x1):

```
// start of compare_pic_by_pic_num_desc
compare_pic_by_pic_num_desc:
  ...
  // Inserted duplicate load, modifying callee-saved
  // register x23 used by library function upon
  // return
  ldr x23, [x24]
  // Original load
  ldr x0, [x1]
  ...
```

Error Manifests as a segmentation fault during execution of library code after nZDC instrumented user code returns because callee-saved registers hold values different from what is expected.

Workaround Workaround by removing the nZDC insertion in code called by library code in LLVM.

Residual False Negative As above, directly limits coverage by not covering some functions.

Proper Fix A proper fix would require re-compiling all library code with the nZDC transformation as well.

Lesson Similar to section 3.3.1, unprotected library code may be the caller, and special consideration should be taken so that the error detection does not break the execution of the library code.

[7]We observed this in omnetpp (which unfortunately still does not work due to exception handling) and h264ref benchmarks. For omnetpp, this happened with many functions that are called before the start of main. For h264ref benchmark, this happened with a function called after main. In both cases, while the problem from section 3.3.1 also exists, the workaround to prevent linked libraries destroying shadows is not enough, unlike many other cases where only section 3.3.1 was a problem.

3.3.3 Exception Handling Breaks Redundant Registers.

Shadow registers get their values copied from the original registers at the beginning of main and after function calls (after a function returns). However, execution jumps over this setup after a function call in the case of exceptions.

Cause nZDC inserts instructions to copy values from original to redundant registers after call instructions so that the checking code works on the caller even if the callee does not have nZDC instrumentation. However, this does not work for exception handling since the execution does not resume from the function return.

Example From omnetpp, in the C++ source code:

```
try {
  // Some work
  ...
  // Function that throws an exception
  checkTimeLimits();
  // Some other work
  ...
// Execution skips other work after exception
}
// Handles exception
catch (...) {
```

Error Manifests as a false-positive error detected during execution when checking for error after the point of exception return (and going through library code for exception handling).

Workaround None, not fixed (hence omnetpp segfault).

Proper Fix A proper fix would require inserting register-copy instructions at the point of exception return or adding nZDC instrumentation to exception-handling library code.

Lesson In addition to function calls, exceptions can also take execution into unprotected library code, and do not come back to user code through function returns. Special consideration should be given to exception handling on top of function calls.

3.3.4 Other Calling-Convention Issues.

In general ISAs, there may not be enough general-purpose registers to reserve for shadowing without breaking the calling convention. For example, there are barely enough registers in AArch64 for nZDC to work. Among the 31 general purpose registers in AArch64, 14 are defined to have special uses such as parameter and result passing, frame pointer, and link register, and cannot be reserved for shadowing without breaking the calling convention. This means that 14 other registers have to be reserved to shadow these registers to protect them. Adding in the two extra registers for conditional branches,

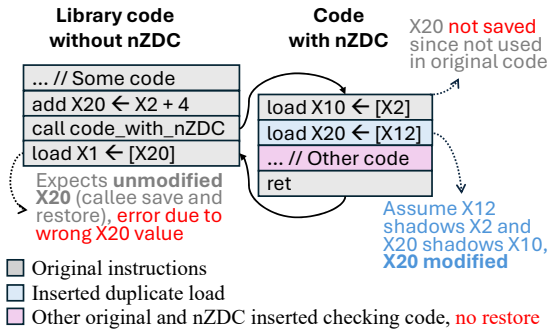

Figure 8. nZDC checking code induces errors by modifying callee-saved register values when called from library code compiled without nZDC transformation.

one for our store-check fix (section 3.1), there are 14 original registers and, 17 reserved registers. While this is just enough in theory, practical issues (section 3.4.3) mean that 2 registers get saved and restored by LLVM currently even when reserved, breaking shadows and reservations, limiting us to just 15 safely reserved registers that are never inadvertently clobbered, which is not enough to implement branch checking with a full set of shadows, though branch checking is currently missing anyway (section 3.2).

3.4 Implementation Oversights

This section describes details and lessons arising from various oversights that arise from the implementation, rather than the fundamental design of nZDC.

One general issue is mishandling of edge cases or instructions: the solution given typically works for the common cases seen in Mibench [12], but some less common but not rare cases need to be handled differently, and occur frequently in large workloads such as SPEC [14].

Another problem considers interaction with other LLVM passes. The additions from the nZDC transformation can benefit from some passes/optimizations and be broken by others. The original paper [6] justifies performing the transformation at a late stage in order to still take advantage of optimizations such as dead code elimination and common sub-expression elimination without them eliminating the inserted error checking code. It also must be done post-register-allocation, in order to maintain duplicate shadow registers with a mapping from each architectural register to its own shadow. However, some passes could have been beneficial to run after nZDC, and some optimizations still come after and should have been taken into consideration.

3.4.1 Missing Labels for Inserted Error Recovery Block.

Cause With LLVM 3.7.1, the label of the error-handling basic blocks inserted at the end of a function is optimised out if the last instruction prior to insertion was not `ret`. This may be due to the late insertion of conditional branches in the backend that fails to split the basic block, and/or missing out on control flow-analysis such as if-conversion. The label is optimised out when the block with inserted conditional branches has the error-handling block as its only successor, and is itself the only predecessor of the error handling block at the same time.

Example From bzip2, in the generated assembly file:

```
// Function originally has only 1 block that ends
// with bl exit
configError:
  // Some work
  ...
  // Inserted conditional branch, block not split
  cbnz x25, .LBB6_1
  bl exit
  // Inserted copy after call
  ...
  sub x23, x0, xzr
// Commented label (.LBB6_1, optimised out)
// BB#1:
  // Body of error recovery block
  ...
```

Error Compilation error during object-file generation such as `undefined reference to '.LBB6_1'`.

Fix Fixed through re-adding the correct label in assembly.

Lesson Insertion of conditional branches changes the control flow of the block they are inserted in. Subsequent optimizations should be made aware of this change.

3.4.2 Duplicate Destinations for Load-Pair Insts.

Cause When checking for store pair instructions, the stored values are re-loaded into the same registers. However, while the store pair can store the value from the same register twice to two memory locations, the load pair cannot load to the same register.

Example From xalancbmk, in the generated assembly file (assumes x24 shadows x1):

```
// Original store pair, w2 → [x1+56] and w2 →
// [x1+60]
stp w2, w2, [x1, #56]
// Inserted checking-load
ldp w2, w2, [x24, #56]
```

Error Manifests as a compilation error such as *Warning: unpredictable load of register pair* – `'ldp w2,w2,[x24,#56]'`

Workaround Workaround by replacing the load pair instruction with a simple load from only the first memory location in assembly.

Residual False Negative Second store of a pair unchecked.

Proper Fix A proper fix would need to add a second load to the same register from the second memory location after the loaded value from the first memory location has been checked.

Lesson Loads are not perfect symmetries of stores, so take extra care wherever loads are intuitively treated as such.

3.4.3 Breaking the Calling Convention.

Cause The original implementation of nZDC reserves registers to be used as shadows even for those with special uses according to the calling convention.

Example In the generated assembly file (constructed example from sjeng and perlbench to show multiple cases at once, assumes x21 shadows sp, x27 shadows x17, x8 shadows x16). The caller sets up x8 to hold the address of the space allocated for the return value that is of compound type before calling, and this conflicts with using x8 as shadow register which should hold the same value as x16.

```
// Original store callee-saved registers
stp x27, x28, [sp, #-48]!
...
// Duplicate add, no substitute to shadow
add x8, x21, #8
// Original add setting up x8 with address for return
// of compound type
add x8, sp, #8
// Checks before call
...
// x16 different from x8 without add
sub x25, x16, x8
// Error detected
cbnz x25, .LBB23_7
...
// Call function which uses x8 for return value
bl think
...
// Setting up x27 to shadow x17 after return
sub x27, x17, xzr
...
// Duplicate sub
sub x27, x27, x1
// Original sub
sub x17, x17, x1
// Duplicate restore, no substitute to shadow
ldp x27, x28, [x21], #48
// Original restore
ldp x27, x28, [sp], #48
// Check before call, restored x27 differ from x17
sub x25, x17, x27
// False positive
cbnz x25, .LBB955_26
b some_other_func
```

Error Manifests as execution failure/output error or as false-positive error detected during execution.

Workaround Workaround by changing which registers are reserved in LLVM. However, this does not completely fix the problem because some registers marked as reserved (for being shadows) that are also callee-saved registers get saved and restored despite the fact they should not have been used for anything other than checking. And restoring a redundant register's value after the original register's value already changed results in discrepancies between the value in the redundant and the original registers. We could also work around redundant register's store/restore problem by removing registers from the list of callee-saved registers, but this breaks the calling convention.

Proper Fix We are not sure why reserved registers are still saved/restored. But making sure reserved registers for shadowing non-callee-saved registers are not restored should solve the problem.

Lesson The calling convention should be taken into consideration when selecting registers to be reserved and used for other purposes.

3.4.4 Conditional Branches with Target Out-of-Range.

Cause nZDC inserts a large number of instructions, which sometimes pushes the branch targets beyond the range of the conditional branch instructions.

Example From gcc, in the generated assembly file:

```
// Original conditional branch to 14573 lines later,
// out of the +/-32KB range
tbnz w8, #5, .LBB1358_421
```

Error Manifests as a compilation error during object file generation such as `Error: conditional branch out of range` with gcc or `fatal error: error in backend: fixup value out of range` with clang.

Workaround Workaround by moving the nZDC pass before the branch relaxation pass in LLVM. Branch relaxation pass replaces out-of-range conditional branches with a direct branch instruction to original taken target with an inserted conditional branch before it that jumps to the original not-taken target with inverted condition. But this does not completely fix the problem because some conditional branches still have targets out-of-range despite putting the nZDC pass before the branch relaxation pass.

Proper Fix We are not sure why some conditional branches with a target out-of-range are not fixed by the branch relaxation pass. But applying the same logic to transform these conditional branches to direct branches should solve the problem. Note that this is in the absence of control-flow checking, which if implemented (section 3.2) would be affected by the branch-relaxation pass due to the changes in conditional branches.

Lesson Late transformation does not affect previous optimizations, but may still need some of those optimizations to work.

3.5 Other Implementation Oversights

We faced several other barriers to successful execution, where the fixes were simpler or with less profound insights, which we summarise briefly here.

Duplicate label inserted, symbol defined twice. The instruction-duplication pass failed to filter out some labels, duplicating them along with other instructions. We added a manual fix to detect duplicated labels and remove the copies, leaving only the duplicated label-free instruction behind.

Incorrect register substitution in some inserted loads. nZDC checks for errors in stores by inserting matching loads, and substitutes the original register holding the memory-access address by the matching redundant register. However, register substitution is incorrect/missing for some inserted load instructions, particularly when used in pre/post-indexed loads (which modify the value in the register used for base address). This results in false positives. We fixed this through correcting the operand indexing on register substitution for pre/post-indexed loads in LLVM.

Incomplete duplication of store instructions. Similar issues appear on pre/post-indexed stores, where the checking load to verify the store is missed out. In addition to false positives, this results in false negatives, as while there is still a check of the shadow, the original's value comes from the previous computation rather checking the value in memory is correct. We added a workaround to reinsert this checking load for pre/post-indexed stores to avoid false positives[8].

Missing register-value copying on floating-point/vector registers. Floating-point(FP)/vector registers also have registers reserved as redundant registers and their values can be used to calculate integer register values. However, the nZDC implementation is missing the register-value copy to set up these registers, presumably because these cases did not appear in Mibench. These manifest as false positives, because the shadows do not match. We fixed with a register-value copy from original to shadow floating-point/vector registers.

3.6 Miscellaneous Problems

This section describes problems not related to nZDC scheme but exists in the repository [19].

[8]Some other store instructions are still missing the following load insertion, resulting in incomplete checking, so false negatives. To do this properly, all the different types of store instructions in the machine instruction representation should have a case in `covertSTtopcodeToLoad` with a corresponding load instruction identified. This requires coverage of an exhaustive list of cases, because stores of different sizes, source operands types, extensions etc. are each a different machine instruction. To get the substitution done properly, for each of those instructions, one would also need to verify what is the index of the source operand with the base memory address.

Cannot return unique_ptr as bool with gcc-12. When trying to compile the repository LLVM [19] with gcc-12, we encountered a type error where a `unique_ptr` type variable is being returned as a `bool` type. This is due to an update in gcc which made this illegal, and the vanilla LLVM 3.7.1 also has the same problem (any old code may).

This is fixed by adding a static cast of the return variable `MDMap` to `bool` before returning it in function `hasMD` from file llvm/include/llvm/IR/ValueMap.h.

Segmentation fault during compilation of assembly. We initially encountered a segmentation fault when compiling with the repository LLVM [19] even without enabling the nZDC options. The LLVM stack dump showed `Running pass 'Machine Instruction Scheduler' on function 'XXX'` with XXX being the first function in the file.

After comparing against the vanilla LLVM 3.7.1 code, we found that this is due to the modified compiler [19] moving member `NumOperands` of class `MachineInstr` to before another member `MCID`, which then somehow resulted in `MCID` becoming null. We fixed this by reverting the position of `NumOperands` to after member `Operands` (which is also after `MCID`), and changing the instance of direct access to `NumOperands` in function `checkStoresnZDC` to using the getter method `getNumOperands`.

4 Working Status on Benchmarks

After fixing or working around the problems found in the implementation [19] for ARMv8 AArch64 ISA, we applied nZDC to more complex and widely used SPEC2006 and PARSEC benchmarks. Table 1 shows the working status of each tested benchmarks, benchmarks not listed are not tested. We have tested all 12 SPEC2006 integer benchmarks using ref input (only the first where multiple exist) and 6 PARSEC benchmarks using simmedium input. We execute on an 80-core AArch64 Neoverse-N1 server with 512GB RAM.

The linked standard libraries used do not have the nZDC transformation applied, only benchmark code compiled from source has nZDC applied. This follows the methodology of the original paper [6], which also did not apply protection to libraries, despite us finding several compatibility challenges needing workarounds (section 3.3). Most tested benchmarks can run to completion with the correct output. Benchmarks gcc and xalancbmk show a compilation error related to conditional branches having out-of-range targets (see our discussion on this problem and our solution attempts in section 3.4.4). Benchmark gobmk runs to completion but has an error in the generated output, due to the transformation incorrectly altering program behaviour. Benchmark omnetpp runs into a segmentation fault due to exception handling taking execution to linked-library code and back without going through the code immediately after function return (see our discussion on this problem in section 3.3.3).

Table 1. Working status of tested benchmarks.

SPEC2006 integer		PARSEC	
Benchmark	Status	Benchmark	Status
perlbench	Works	blackscholes	Works
bzip2	Works	dedup	Works
gcc	Compilation error	fluidanimate	Works
mcf	Works	freqmine	Works
gobmk	Output error	streamcluster	Works
hmmer	Works	swaptions	Works
sjeng	Works		
libquantum	Works		
h264ref	Works		
omnetpp	Segmentation fault		
astar	Works		
xalancbmk	Compilation error		

Figure 9 shows the slowdown of nZDC on the working benchmarks compared to the execution of a binary without nZDC. The baseline binary without nZDC is compiled with vanilla LLVM version 3.7.1, which is the version of LLVM that the nZDC implementation [19] is based on[9]. The overheads of SPEC CPU2006, at just over 2× geomean, are comparable to the original paper's [6] Mibench average of 2.1×. PARSEC sees slightly lower overheads, perhaps due to an increased proportion of floating-point instructions[10]

5 Related Work

Software error detection techniques have used redundancy at different levels. nZDC [6] is a software error detection techniques using instruction-level duplication. One of the first works in this area was EDDI [24] where all instructions including stores are duplicated, which resulted in not only high performance overhead but also doubled the memory footprint. SWIFT [26] improved upon EDDI by assuming that the memory subsystem is protected with ECC and does not duplicate stores, removing the memory overhead. nZDC improves on the vulnerabilities of SWIFT, and follow up works [7, 9] improve on the limitations of nZDC. To avoid the high performance overhead and still keep high (but not complete) coverage, some works [11, 22] leverage the flexibility of instruction-level duplication to perform selective instruction duplication where only some critical instructions that are likely to generate errors are duplicated.

Other techniques use thread/process-level duplication [17, 20, 23, 25, 27, 30–32], where the duplicate execution happens on a different thread/process instead of being duplicate instructions inserted sequentially. The redundant threads typically run in parallel to the original execution, showing lower performance overhead compared to instruction-level duplication and have their own set of architectural registers. However, the check is also at a more coarse granularity without the flexibility of instruction-level duplication and with longer detection delay.

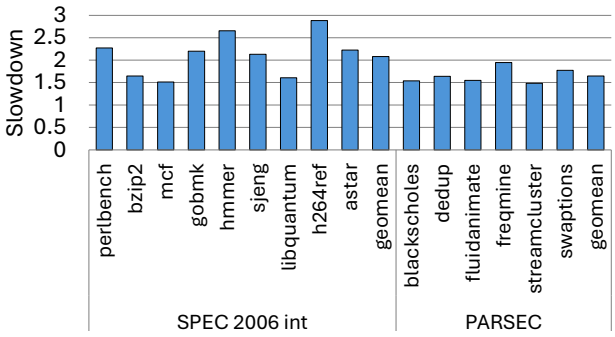

Figure 9. Slowdown of benchmarks with nZDC applied compared to a baseline without nZDC.

6 Conclusion

This paper has analysed the challenges that the only current open-source software error detection system, nZDC, faces when applied at the scale of major benchmarks. We hope that it serves to inspire new research in this field that is rapidly taking on increased relevance due to widespread reports of real-world faults [4, 10, 15]. To be clear, the fixes we generate are not a complete solution, nor do we intend to produce one in future: our goal here was initially to use this technique as a comparator technique for a rival mechanism in a very different part of the design space. However, we thought the things we found out in doing so were too interesting to not give proper analytical treatment. Our modified nZDC is sufficient for performance analysis, e.g. to use as a target to benchmark against, for new researchers in both hardware and software-based fault tolerance to compare against and improve. We indicate in many places where false negatives and false positives are not entirely fixed, and hope these and our hints at solutions provide ample ideas for future compiler researchers and engineers to provide more comprehensive solutions, as well as valuable insights into pitfalls when working in fault tolerance more generally.

[9]The compiler pass as stands would require non-trivial porting effort to the latest LLVM. However, users of our extensions can also use version 3.7.1 to do reruns of nZDC as we have. We also believe that the insights here will be just as valuable to anyone porting this codebase to future versions; none of the issues we uncover are particularly related to assumptions in LLVM 3.7 (as they are ISA- or ABI-related, not compiler-related).

[10]The overheads here are intended as a lower bound; true overheads may be slightly higher due to coverage checks not completely fixed, since our goal was to get the technique up and running. 1) As in the original work [6], standard libraries linked to our nZDC binaries do not have nZDC instrumentation, and while SPEC spends negligible time in libraries, PARSEC is in libraries for around 10% of baseline execution. 2) Conditional-branch checking is missing (section 3.2), and 3) the checking load insertion may still be missing on some stores (as explained in section 3.5, we have only ensured that the pre/post-indexed ones have the checking load).

References

[1] Arm Ltd. 2023. *Arm Cortex-X2 Core Technical Reference Manual. Cache protection behavior.* https://developer.arm.com/documentation/101803/0200/RAS-Extension-support-/Cache-protection-behavior

[2] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. 2003. Fault-Tolerant Platforms for Automotive Safety-Critical Applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*. doi:10.1145/951710.951734

[3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph. D. Dissertation. Princeton University.

[4] Matthew Connatser. 2024. Game dev accuses Intel of selling 'defective' Raptor Lake CPUs. https://www.theregister.com/2024/07/13/game_raptor_intel/.

[5] Moslem Didehban, Sai Ram Dheeraj Lokam, and Aviral Shrivastava. 2017. InCheck: An In-Application Recovery Scheme for Soft Errors. In *Proceedings of the 54th Annual Design Automation Conference (DAC '17)*. Article 40. doi:10.1145/3061639.3062265

[6] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: A Compiler Technique for near Zero Silent Data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. Article 48. doi:10.1145/2897937.2898054

[7] Moslem Didehban and Aviral Shrivastava. 2018. A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments. *IEEE Transactions on Reliability* 67, 1 (2018). doi:10.1109/TR.2018.2793098

[8] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. 2017. NEMESIS: a software approach for computing in presence of soft errors. In *Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD '17)*. doi:10.1109/ICCAD.2017.8203792

[9] Moslem Didehban, Hwisoo So, Prudhvi Gali, Aviral Shrivastava, and Kyoungwoo Lee. 2024. Generic Soft Error Data and Control Flow Error Detection by Instruction Duplication. *IEEE Transactions on Dependable and Secure Computing* 21, 1 (2024). doi:10.1109/TDSC.2023.3245842

[10] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. *CoRR* abs/2102.11245 (2021). arXiv:2102.11245 https://arxiv.org/abs/2102.11245

[11] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. doi:10.1145/1735970.1736063

[12] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization (WWC-4)*. doi:10.1109/WWC.2001.990739

[13] Daniel Henderson and Irving Baysah. 2021. Introduction to IBM Power® Reliability, Availability, and Serviceability for POWER9® processor-based systems using IBM PowerVM™ with updates covering the latest Power10 processor-based systems. https://www.ibm.com/downloads/cas/2RJYYJML.

[14] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006). doi:10.1145/1186736.1186737

[15] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores That Don't Count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. doi:10.1145/3458336.3465297

[16] Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. 2019. The Arm Triple Core Lock-Step (TCLS) Processor. *ACM Trans. Comput. Syst.* 36, 3 (June 2019). doi:10.1145/3323917

[17] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M. Jones. 2016. COMET: communication-optimised multi-threaded error-detection technique. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '16)*. doi:10.1145/2968455.2968508

[18] Moslem2179. [n. d.]. LLVM-FT-mor1kx. https://github.com/Moslem2179/LLVM-FT-mor1kx.

[19] MPSLab-ASU. [n. d.]. nZDC-Compiler. https://github.com/MPSLab-ASU/nZDC-Compiler.

[20] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. 2002. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. doi:10.1109/ISCA.2002.1003566

[21] Khang T Nguyen. 2017. New Reliability, Availability, and Serviceability (RAS) Features in the Intel Xeon Processor Family. https://www.intel.com/content/www/us/en/developer/articles/technical/new-reliability-availability-and-serviceability-ras-features-in-the-intel-xeon-processor.html.

[22] Xiang Ni and Laxmikant V. Kale. 2016. FlipBack: Automatic Targeted Protection against Silent Data Corruption. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. doi:10.1109/SC.2016.28

[23] Shuou Nomura, Matthew D. Sinclair, Chen-Han Ho, Venkatraman Govindaraju, Marc de Kruijf, and Karthikeyan Sankaralingam. 2011. Sampling + DMR: Practical and low-overhead permanent fault detection. In *38th Annual International Symposium on Computer Architecture (ISCA)*. doi:10.1145/2024723.2000089

[24] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002). doi:10.1109/24.994913

[25] S.K. Reinhardt and S.S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. doi:10.1145/342001.339652

[26] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. doi:10.1109/CGO.2005.34

[27] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. 2007. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*. doi:10.1109/DSN.2007.98

[28] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. 2018. EXPERT: Effective and flexible error protection by redundant multithreading. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. doi:10.23919/DATE.2018.8342065

[29] Moor Insights & Strategy. 2017. *AMD EPYC brings new RAS Capability.* Technical Report. https://www.amd.com/system/files/2017-06/AMD-EPYC-Brings-New-RAS-Capability.pdf

[30] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. 2007. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization (CGO '07)*. doi:10.1109/CGO.2007.7

[31] Boyue Zhang, Sam Ainsworth, Lev Mukhanov, and Timothy M. Jones. 2025. Parallaft: Runtime-based CPU Fault Tolerance via Heterogeneous Parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '25)*. doi:10.1145/3696443.3708946

[32] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Runtime Asynchronous Fault Tolerance via Speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. doi:10.1145/2259016.2259035

[33] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. 2010. DAFT: Decoupled acyclic fault tolerance. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT).*

doi:10.1145/1854273.1854289

Received 2024-11-12; accepted 2024-12-21