Comprehensive Backend Support for Local Memory Fault Tolerance
Comprehensive Backend Support for Local Memory Fault Tolerance

Norman A. Rink Jeronimo Castrillon
Center for Advancing Electronics Dresden
Technische Universit"at Dresden, Germany

Abstract
Technological advances drive hardware to ever smaller feature sizes, causing devices to become more vulnerable to transient faults. Applications can be protected against faults by adding error detection and recovery measures in software. This is popularly achieved by applying automatic program transformations. However, transformations applied to program representations at abstraction levels higher than machine instructions are fundamentally incapable of protecting against vulnerabilities that are introduced during compilation. In particular, a large proportion of a program’s memory accesses are introduced by the compiler backend. This report presents a backend that protects these accesses against faults in the memory system. It is demonstrated that the presented backend can detect all single bit flips in memory that would be missed by an error detection scheme that operates on the LLVM intermediate representation of programs. The presented compiler backend is obtained by modifying the LLVM backend for the x86 architecture. On a subset of SPEC CINT2006 the runtime overhead incurred by the backend modifications amounts to 1.50x for the 32-bit processor architecture i386, and 1.13x for the 64-bit architecture x86_64.

To achieve comprehensive detection of memory faults, the modified backend implements an adjusted calling convention that leaves library function calls transparent and intact.

Keywords transient hardware faults, soft errors, memory faults, error detection, fault tolerance, resilience, compiler backend, code generation, intermediate representation (IR), LLVM

1. Introduction
As a result of aggressive technology scaling, transient hardware faults occur at increasing rates [3, 5, 9, 45]. Systematic studies have found that faults lead to erroneous application behavior with non-negligible probabilities [23, 34, 43], and it is known that transient faults can have consequences that are as dramatic as entire system outages [1]. Although transient hardware faults, also known as soft errors, are most commonly attributed to charge generation by cosmic radiation [3], shrinking feature sizes increase the vulnerability of devices to general variations in the operating environment, such as variations in supply voltage and temperature [9, 44]. Moreover, tightening temperature budgets may force devices to operate at near-threshold voltage, which reduces reliability [15, 44, 48]. Similarly, to reduce the energy consumption of memory modules, operating voltages can be lowered for SRAM [16] and refresh cycles can be extended for DRAM [31, 50], both of which negatively affect the reliability of data retention.

Although fault rates in individual devices are low, the possibility of faults poses serious problems for small-scale and large-scale computing applications alike. Undetected faults in embedded-devices, as used in safety-critical applications in the automotive or aerospace domain, can pose a danger to human life. At the other end of the spectrum, large-scale services, like those provided by data centers, suffer noticeably from hardware faults since faulting probabilities compound across the large numbers of computers in data centers [1, 23, 34, 43]. Therefore, software that is designed for applications with strict safety and reliability requirements must incorporate measures to tolerate transient hardware faults.

Software can be made fault-tolerant by adding integrity checks to program code. When a check fails, an error has been detected and suitable measures can be taken to recover from it. To enable checks, and hence error detection, some form of redundancy must be added to programs. This can be done conveniently by applying automatic program transformations, such as source-to-source transformations, cf. [28, 38]. With the rising popularity of the LLVM framework and intermediate representation (IR) [30], many fault tolerance schemes have appeared that are implemented as IR transformations, e.g. [12, 17–19, 29, 40, 42, 52]. Operating on IR has the advantages of target-independence and increased productivity compared with operating on machine instructions. However, when transformations are applied to programs at an abstraction level above machine instructions, the compiler backend may introduce new vulnerabilities to faults. Specifically, the backend introduces numerous additional memory accesses. Figure 1 shows the percentages of dynamic load operations that originate from load instructions present in the IR of twelve test programs, labeled A–L:1; there is always some proportion of loads that are inserted by the backend, and in extreme situations (H, L) none or hardly any of the loads appear in the IR. As a consequence, comprehensive detection of memory faults cannot be achieved by fault tolerance schemes that are implemented at the IR level.

1The test programs are introduced in Section 4, cf. Table 1.
This report presents a compiler backend for the C programming language that supports IR-based fault tolerance schemes in detecting all errors that result from faults in the memory system. The backend implements error detection by dual modular redundancy (DMR), i.e., by duplicating memory accesses. Since the backend only inserts accesses to local memory, almost exclusively to the local program stack, duplication poses no issues for multi-threaded programs. It is demonstrated that the presented backend can indeed support an IR-based error detection mechanism in detecting all memory faults that result in single bit flips. In fact, the presented backend’s capability to detect errors goes beyond single bit flips since DMR can generally detect any number of corrupted bits in a single data word. Moreover, when multiple bit flips affect more than one data word, there is still a high probability that this can be detected, especially if the redundant copies of the same data word are not stored at adjacent positions in memory. The modifications required to implement DMR for memory accesses have been added to the LLVM backend for the x86 architecture [30]. It should be stressed that the modified compiler backend can be combined with arbitrary error detection schemes at the IR level.

Previous work has implemented entire fault tolerance schemes by modifying compiler backends [14, 33, 37, 39, 51], but it is usually assumed that memory is protected by hardware measures, such as ECC. The present report does not make this assumption since cost considerations may rule out using ECC memory at all levels in the memory hierarchy. Specifically, ECC memory introduces an area overhead that may be unacceptable for on-chip components of the memory system, such as low cache levels or load-store queues [14]. For a system to be vulnerable to memory faults, it suffices that there is a single unprotected component in the memory hierarchy. On such vulnerable systems, comprehensive error detection can still be implemented in software, e.g., by the methods presented in this report. This is particularly useful when the safety and reliability requirements change during the lifetime of a system. Moreover, the presented approach to error detection combines backend modifications with an IR-based mechanism, which is more flexible and less target-dependent than fault tolerance schemes that are implemented entirely in a compiler backend.

This report is structured as follows. Section 2 introduces memory faults and identifies the vulnerable memory accesses that are inserted by the compiler backend. Section 3 explains in detail how our backend modifications are implemented. Section 4 introduces the suite of test programs that are used to demonstrate the effectiveness of our error detection scheme. Results are presented in Section 5. Section 6 discusses related work, and Section 7 summarizes and discusses the findings of this report.

2. Background and Motivation

Many fault tolerance schemes have been implemented at the level of LLVM IR, e.g., [12, 17–19, 29, 40, 42, 52]. These approaches accept the fact that the convenience of operating on target-independent IR comes at the price of losing some amount of control over the generated machine code and its vulnerability to faults. It is indeed known that compiler optimizations can affect fault tolerance levels [13, 18]. Sometimes relaxed error detection rates are even desirable if this leads to reduced runtime overheads [17, 28, 40]. However, whenever trade-offs between fault tolerance and overheads are exploited, developers may not want to be at the whim of the compiler backend. Instead, when error detection or recovery mechanisms are implemented at a certain level of abstraction, developers should be guaranteed that subsequent steps in the compilation process will not introduce new vulnerabilities to faults that have already been addressed by the implemented mechanism. This report presents a compiler backend that meets this requirement for schemes that detect faults in the memory system, including the load-store queue, caches, and communication buses.

![Fig. 2: Liveness of data in memory, with pointers p and q.](image)

Fig. 2: Liveness of data in memory, with pointers p and q. p ≠ q.

2.1 Memory faults

Typical faults in memory cells are bit flips caused by energetic particles that originate from cosmic radiation [3]. However, motivated by the current trend toward reducing power consumption, it has been suggested that the operating voltage of SRAM be lowered [16], and that refresh cycles of DRAM modules be extended [31, 50]. Both suggestions reduce the capability to retain data and hence increase the probability of memory faults.

The probability that a data word is corrupted by a fault increases with the time that the data word spends in memory. Figure 2 shows two pairs of store and load operations in a program’s instruction stream. Memory is accessed at addresses p and q, p ≠ q, and it is assumed that there are no other accesses at these addresses. The data word at p has a much longer lifetime and hence is more likely to become corrupted than the data word at q. Therefore, when considering fault tolerance measures for the memory system, it is reasonable to target main memory first since this is where the lifetimes of data will generally be the longest. This also means that when, say, ECC are implemented in hardware, on-chip memories, such as low cache levels or load-store queues, may not be protected. In fact, the need to protect load-store queues has recently been stressed [14].

The backend presented in this report is intended not to leave any memory accesses that it inserts vulnerable to faults. Therefore, error detection must be applied to all memory accesses, regardless of the lifetimes of data in memory. In particular, this comprehensive error detection strategy also serves to protect data that never leaves on-chip memories, which, according to the previous paragraph, are less likely to be protected against faults by hardware measures.

2.2 Approaches to error detection

Error detection schemes work by maintaining redundant information that is used to check the integrity of data. This is most evident in DMR-based error detection schemes, where two copies are kept of each data word. Errors can then be detected by comparing these copies. If the two copies disagree, an error must have occurred in at least one of them. In this way, all single bit flips can be detected. Multiple bit flips can also be detected, provided they do not affect the two copies in identical ways. In particular, multiple bit flips can always be detected if they occur in only one of the copies.

Protecting data in memory by DMR can be problematic in the context of multi-threaded applications since care must be taken to avoid race conditions when different threads access redundant copies of data. In this report, DMR-based error detection is applied selectively only to those memory accesses that are introduced by the compiler backend. Since these accesses are local, no race conditions can result in multi-threaded applications.

An alternative approach to error detection is based on encoding data: if the set of valid code words is a small subset of all possible data words, a hardware fault is likely to produce a data word that is not also a valid code word. Hence, errors can be detected by checking whether data words are also valid code words. When data is encoded, additional bits are typically required to represent code...
words. Although these bits contain redundant information, no data is duplicated explicitly. Therefore, encoding-based schemes can immediately be applied to multi-threaded applications.

A simple, yet effective, encoding-based error detection scheme for integer values can be defined by decreeing that the valid code words are precisely the multiples of a fixed integer constant \( A \). This is known as AN encoding \([10, 20]\). To enable the detection of errors specifically in the memory system, an integer value \( m \) must be encoded before being stored:

\[
m_{\text{encoded}} = m \cdot A.
\]

Consequently, whenever a value \( m_{\text{encoded}} \) is loaded from memory, it must be decoded before further processing takes place:

\[
m = m_{\text{encoded}} / A.
\]

Errors can be detected by evaluating the following boolean expression for a value \( n \) that has been loaded from memory:

\[
n \mod A = 0.
\]

In the absence of memory faults, the value \( n \) is a valid code word. Hence, if expression (1) evaluates to FALSE, a fault must have occurred.

2.3 AN encoding at the level of intermediate representation

We have implemented the AN encoding scheme from Section 2.2 as a program transformation that operates on LLVM IR. The transformation instruments store and load instructions with multiplication and division respectively, as shown in Listing 1, to facilitate encoding and decoding. Checking is performed immediately after load instructions. If the check fails, the program exits with the special exit code ENCODING, which indicates that an error has been detected by the AN encoding scheme.

Listing 1: Store and load instructions

```c
some_bb:
    ...%1 = mul i64 %0, %a ; encode
    store i64 %1, i64* %p
    ...%2 = load i64* %p
%3 = srem i64 %2, %a
%4 = icmp eq i64 %3, 0 ; check
br ii %4, label %next_bb, label %exit_bb
next_bb:
%5 = sdiv i64 %2, %a ; decode...
```

exit_bb:
call void @exit(132 ENCODING)

As evidenced by Figure 1, one cannot expect that all errors resulting from memory faults are detected if the AN encoding scheme is applied at the IR level. The IR is lowered to machine representation instruments store and load instructions with multiplication and division respectively, as shown in Listing 1, to facilitate encoding and decoding. Checking is performed immediately after load instructions. If the check fails, the program exits with the special exit code ENCODING, which indicates that an error has been detected by the AN encoding scheme.

3. Backend Implementation Details

We have modified the LLVM backend \([30]\) for the x86 architecture to implement DMR-based error detection for the memory accesses listed at the beginning of Section 2.4. This means that whenever a data word is written to memory, a second copy of the same data word is also stored. When the data word is re-loaded, the two copies are compared. Disagreement between the two copies indicates the presence of an error caused by a fault in the memory system.

Following the detection of an error, suitable recovery measures can be taken. This report concentrates on error detection only. Hence, upon detecting an error in a memory access that has been inserted by the backend, the executing program is terminated with the special exit code BACKEND.

In the following, 32-bit machine code is used to illustrate implementation details. The corresponding 64-bit machine code uses 64-bit registers but is otherwise identical to the 32-bit code.

Listing 2: CJE instruction.

```c
...mov −0x30(ebp), ecx
CJE −0x34(ebp), ecx
add ecx, esi
...`

Listing 3: CJE expansion.

```c
...mov −0x30(ebp), ecx
CJE −0x34(ebp), ecx
cmp −0x34(ebp), ecx
jne <exit>
add ecx, esi
...`

Listing 4: Live flags register.

```c
mov −0x30(ebp), ecx
lahf
cmp −0x34(ebp), ecx
jne <exit>
sahf
add ecx, esi
...`

Listing 5: Live flags and eax.

```c
mov −0x30(ebp), eax
xchg eax, ebx
lahf
xchg eax, ebx
cmp −0x34(ebp), eax
jne <exit>
xchg eax, ebx
sahf
xchg eax, ebx
add eax, esi
...`
3.1 The CJE pseudo-instruction

To facilitate error detection, the pseudo-instruction CJE (compare and jump to exit) has been introduced. After loading a value into a register, the CJE instruction is used to compare the register with the second copy of the value in memory. If the comparison fails, a jump to an exit sequence is performed. The CJE instruction is expanded into native machine instructions at the very end of the compilation process, immediately before the emission of machine code. A typical occurrence and the expansion of the CJE instruction are shown in Listings 2 and 3 respectively, where it is assumed that two copies of the same value are stored at offsets -0x30 and -0x34 from the frame pointer (in the ebp register).

Expanding CJE introduces a cmp instruction, which means that the flags register is overwritten. Therefore, if the flags register is live at the CJE instruction, its contents must be saved. Since CJE expansion happens late in the compilation process, registers have been allocated and the liveness of the flags register can easily be determined. To save the contents of the flags register, it is preferable not to write to memory as this would introduce a new vulnerability. The only x86 instructions that transfer the flags register to and from a general purpose register are the lahfd and sahf instructions respectively, which use the ah register. Listing 4 shows the resulting expansion of CJE if the flags register is live. Finally, if the eax register is also live at the CJE instruction, as in Listing 5, its contents too must be saved and restored around the lahfd and sahf instructions. For this purpose we reserve the ebx register. In between the lahfd and sahf instructions in Listing 5, the ebx register contains the saved value of the flags register.

It may seem like a drastic step to reserve a register solely for handling CJE expansion, especially on the 32-bit x86 architecture, which has only eight general purpose registers. However, it is explained in Section 3.5 that the ebx register must be reserved to pass the return address to called functions. This register can then also be used for handling the CJE expansion in Listing 5, which therefore creates no additional register pressure.

3.2 Register spills (spill)

Memory faults can be detected in spilled values by spilling registers to two memory locations. Therefore, whenever the register allocator introduces a spill, a pair of spill slots is allocated on the stack, and the spilled value is stored to both slots. When the value is restored from the first spill slot to a register, a CJE instruction is inserted before the next use of the register. Listing 6 shows a typical register spill and subsequent restore without error detection. Listing 7 shows the spill and restore code with DMR-based error detection.

To detect faults that affect the values of callee-saved registers while they reside on the stack, one could of course push every register onto the stack twice. However, this would require individual CJE instructions for each register when values are restored from the stack. Instead, we compute the running sum of the values in callee-saved registers as they are being pushed onto the stack. When all callee-saved registers have been processed, the final sum is also pushed onto the stack. The top half of Listing 9 illustrates this, where the sum is computed in edi. Before the callee-saved registers are restored, their sum is first popped off the stack; values that are subsequently restored to registers are then subtracted from the sum. When it comes to popping the final callee-saved register off the stack, the sum has been reduced to the remaining value on the stack. A single CJE instruction checks that this is indeed the case, as shown in the bottom half of Listing 9. After a successful check, the final register need not be popped of the stack: after the CJE instruction in Listing 9, the register edi already contains the value to which it must be restored. Therefore, all that is left to do is to increment the stack pointer. In fact, if instead of incrementing the stack pointer, another pop instruction were performed, this would constitute another memory access, and hence a vulnerability.

Note that the CJE instruction that is inserted after popping callee-saved registers off the stack is placed in the function return sequence. Since the flags register is not live at this point, the CJE expansion from Listing 3 can always be applied.

3.3 Callee-saved registers (csr)

Callee-saved registers are pushed onto the stack immediately after function entry, and are popped off the stack immediately before the function returns. Typical instruction sequences for this are shown in Listing 8, with the callee-saved registers edi and esi.

3.4 Frame pointer (fptr)

When a function uses the frame pointer, the value of the frame pointer of the enclosing function (in the ebp register) is pushed onto the stack at function entry. The frame pointer is restored to its old value by popping it off the stack immediately before returning. This is illustrated in Listing 10.

Detection of errors that affect the frame pointer is completely analogous to callee-saved registers: a second copy of the old frame pointer is pushed onto the stack at function entry, and when the old frame pointer is restored, the two copies are compared by means of a CJE instruction, as in Listing 11. Again, the flags register is not live at the CJE instruction.

Listing 8: Callee-saved registers.

```
Listing 9: Protection of callee-saved registers.
push edi
push esi
... pop esi
pop edi
ret
```

Listing 10: Standard handling of the frame pointer.

```
Listing 11: Duplication of the frame pointer.
push ebp
mov esp, ebp
... pop ebp
ret
```

4
3.5 Return address (return)

On the x86 architecture, the return address is always passed on the stack. Thus, given the possibility of memory faults, it can never be assumed that the return address is correct. To obtain a copy of the return address that is guaranteed to be correct, even in the presence of memory faults, the calling convention must be modified so that the return address is passed in a register. We reserve register ebx for this purpose. In principle, one could use one of the caller-saved registers eax, edx, or even eax. However, the fastcc calling convention allows that function arguments be passed in these registers, and since the fastcc convention is used frequently, we prefer to leave fastcc unmodified.

A further complication on the x86 architecture is that the return address is not immediately accessible outside of the called function. Therefore, to pass the return address in ebx, the compiler backend must generate code as in Listing 12, where the address of the instruction following the function call appears explicitly as an immediate value.

Listing 12: Passing the return address in ebx.

```
0x804a99e:    mov 0x804a9a8,ebx
0x804a9a3:    call <printf>
0x804a9a8:    mov eax,-0x2c(esp)
```

The first instruction in the called function pushes ebx onto the stack, as done in Listing 13. This means that the two copies of the return address now reside side-by-side on the stack. When the called function returns, error checking of the return address is carried out completely analogously to callee-saved registers. The only subtlety is that the final ret instruction must not be executed since it reads the return address from the stack, which constitutes a vulnerability. Instead, an indirect jump to the checked return address is performed, cf. Listing 13. Once again the flags register is not live during a function’s return sequence, allowing the CJE expansion from Listing 3 to be used. This is particularly fortunate given that the eax register is live during the return sequence if the function returns a value.

Listing 13: Protected function return sequence.

```
push ebx
```

A few things concerning the reserved ebx register are worth pointing out. First, since the called function immediately pushes ebx onto the stack, it can safely be used inside the function to pass return addresses in nested function calls. Moreover, for the same reason, the ebx register can be used as in Listing 5 to store the flags register when both the flags register and eax are live during a CJE instruction. Lastly, on architectures with a designated return register, e.g. on ARM or MIPS processors, protecting the return address against memory faults does not require that an additional register be reserved or that the calling convention be modified.

3.6 Function arguments (arg)

When function arguments are passed on the stack, they are, of course, vulnerable to faults in memory. To detect errors in function arguments, the calling convention has been modified so that a duplicated copy of the sequence of arguments is put on the stack immediately after the original sequence. Whenever one of the original arguments is loaded into a register inside the callee, a CJE instruction compares the value in the register with the corresponding argument in the duplicated sequence of arguments, cf. Figure 3. This is completely analogous to the bottom half of Listing 7, except that offsets relative to the frame pointer are positive for function arguments.

Note that, since function arguments may be loaded from the stack at any point during execution of the callee, the flags register and eax may generally be live at CJE instructions that check for errors in function arguments.

3.7 Calling conventions and library functions

The standard calling convention on x86 has been modified in two ways. First, the return address is passed in the register ebx, in addition to being put on the stack by the call instruction. Second, a duplicated sequence of stack arguments resides on the stack immediately above the original sequence of stack arguments. Note that the obligation to implement this calling convention rests entirely with the caller. This means that, if a callee chooses not to perform error detection on the return address or on its arguments passed on the stack, this does not break function calls. In particular, library functions can still be called fully transparently from within protected functions. However, calls in the opposite direction do not work: when a protected function is called from an unprotected environment, neither the ebx register nor the stack will be set up according to our modified calling convention. Hence the execution of the unprotected function will lead to premature program termination with exit code BACKEND.

3.8 Jump tables (jr)

Jump tables are an efficient way of implementing switch statements [27]. A jump table is an array of addresses of basic blocks. Unlike all the previously discussed vulnerabilities, jump tables do not reside on the stack, but in the code segment. In Listing 14 an index into a jump table has been calculated in register edi. The jump table itself resides at address 0x8048b84. To protect jump tables against errors, the compiler backend duplicates each jump table in the code segment. The example from Listing 14 is then replaced with the code in Listing 15, where the duplicated jump table is placed at address 0x80489c9. Note that, instead of jumping directly to the address stored at the given index in the jump table, an indirect jump is used in Listing 15 to avoid another vulnerable memory access. Also note again that the flags register and eax may generally be live at CJE instructions that protect jumps to addresses kept in jump tables.

3.9 Final notes on implementation details

Implementing the backend modifications that have been discussed in this section required approximately 1500 additional lines of code relative to LLVM 3.5. The 1500 lines of code are distributed across 30 source files. Given that error detection is clearly a cross-cutting concern [6, 26], the use of aspects [47] would be warranted to...
improve software design. However, the source code of the LLVM framework relies heavily on C++ templates, which are not handled by current aspect compilers [46].

4. Test Programs and Code Generation

To demonstrate that combining AN encoding at the IR level with the presented backend modifications succeeds at detecting all single bit flips in memory, the test programs in Table 1 have been subjected to faults. Due to simple combinatorics, the spaces of all possible faults that can affect a program are quite large. Therefore, conducting exhaustive fault experiments is a processor-bound task, which, for the relatively small programs in Table 1, can still be carried out in reasonable time.

Some of the test programs (C, E, K) appear in the MiBench suite [24], and similar programs are often used to evaluate fault tolerance schemes [14, 18, 36–38, 40]. The programs represent typical algorithmic tasks, such as sorting, tree and graph traversal, manipulation of bit patterns, and linear algebra. Test program L consists of a switch statement that selects one of many arguments of the enclosing function. The reason for including this test is that it is the only one that passes function arguments on the stack for the 64-bit calling convention on x86.

The test programs have been evaluated on the i86 architecture (the 32-bit version of x86) and on x86_64 (the 64-bit version of x86). Properties of the binaries generated for these architectures are listed in Tables 2 and 3. The listed properties are: dynamically executed instructions (instr.), dynamically executed load operations (ld.), and the number of dynamic loads, and for a 64-bit binary it is, of course, 64 times the number of loads. Subsequently, the targeted binary is executed. On the i386 architecture this problem does not occur since pointers are only 48 bits wide, and the chosen A fits into 16 bits.

The runtime overheads introduced by the backend modifications from Section 3 have been assessed on the test programs from Table 1, and also on a subset of SPEC CINT2006. Since the modified backend protects programs written in the C language, the following benchmarks, written in C++, were not used for assessment: 471.omnetpp, 473.aatart, and 483.xalanbank. The benchmarks 403.gcc, 456.hammer, and 464.h264ref break the modified calling convention and can therefore not be used in assessing runtime overheads. For example, 464.h264ref uses the qsort library function, to which it passes a comparator function. When the comparator is called from within the library, our modified calling convention is not observed. Hence this benchmark always exits prematurely, with exit code 1.

5. Evaluation

Faults occur rarely in individual devices. Therefore, one must actively inject faults into systems or programs to evaluate the effectiveness of fault tolerance schemes. The error detection mechanisms introduced in this report have been evaluated by symptom-based fault injection [4, 28, 41]. This means that, instead of simulating a fault at the circuit level, the resulting symptom, as seen by the executing program, is modeled. A fault in the memory system results in the corruption of the data word returned by a load operation. This symptom has been injected into executions of the test programs from Table 1, the detailed procedure for which is described in the next section.

5.1 Fault injection experiments

It is common to evaluate error detection schemes by injecting single bit flips, cf. [14, 17, 51]. Therefore, in this report, the symptom of a memory fault is modeled by flipping a single bit in the result of a load operation. To inject this symptom into an executing program, the Intel Pin tool [32] for dynamic program instrumentation has been used. In a first golden run, the targeted binary is executed under the control of the Pin tool, and all dynamic load operations are recorded. Based on this, all possible symptoms are determined. For a 32-bit binary the number of symptoms is equal to 32 times the number of dynamic loads, and for a 64-bit binary it is, of course, 64 times the number of loads. Subsequently, the targeted binary is
executed once for each symptom, and the program’s response to the injected symptom is recorded. A fault injection experiment is a single execution of the targeted binary with an injected symptom.

The outcome of a fault injection experiment is determined by the program’s response to the injected fault, and responses are classified into the following categories:

1. **correct**: Despite the fault, the program has terminated normally and produced correct output.
2. **hang**: If the program runs for longer than 10x its normal execution time, it is deemed to hang and hence is terminated. In practice, e.g. in safety-critical embedded applications, a hardware watchdog may terminate and restart long-running programs.
3. **crash**: The program has terminated abnormally. Either the operating system has terminated the program, e.g. due to a segmentation fault, or the program itself has exited prematurely due to an error condition caused by invalid data.
4. **sdc**: Silent data corruption occurs when the program has terminated normally but has produced incorrect output.
5. **encoding**: The fault has been detected by AN encoding and hence the program has exited with code `EXIT.terminate`.
6. **backend**: The fault has been detected by one of the DMR-based measures introduced by the backend. Hence the program has exited with code `BACKEND`.

Of course, the responses `encoding` and `backend` only occur if AN encoding has been applied to the targeted binary and if the backend modifications from Section 3 have been used respectively.

### 5.2 No error detection

Figure 4 details how the test programs respond to single bit flips in memory when no error detection measures are applied. While abnormal program termination indicates that something has gone wrong, when `sdc` occurs in practice, one has no reason to believe that the computed output is incorrect. Therefore, one is often particularly interested in the proportion of `sdc` [14, 28, 42]. For the 32-bit binaries the proportion of `sdc` is generally larger than for the 64-bit binaries. This is to be expected given the lower number of registers in the 32-bit `i386` architecture: more data words that are relevant for the program output will, at least temporarily, reside in memory and hence be vulnerable to faults.

### 5.3 AN encoding

When AN encoding is applied, the total number of load operations generally increases (cf. the blocks labeled `encoded` in Tables 2 and 3). However, symptoms that manifest themselves in load operations that are already present in the IR will be detected by the AN encoding scheme from Section 2.3. Figure 5 summarizes the program responses to faults in memory when AN encoding is applied at the IR level. Noticeable proportions of `sdc` remain for the 32-bit binaries, but hardly any `sdc` occurs for 64 bits.

Generally, the effectiveness of AN encoding is much higher for the 64-bit binaries. This can again be explained by the larger number of registers in the `x86_64` architecture and the usage of registers for argument passing. Because of this there is lower register pressure, and hence there will be fewer occasions where the backend has to insert additional memory accesses. In other words, a higher proportion of the executed load operations are already present in the IR, and hence can be protected against faults by the AN encoding scheme from Section 2.3.

### 5.4 AN encoding with backend support

When AN encoding at the IR level is combined with the error detection measures inserted by our backend, all single bit flips in memory are detected, as evidenced by Figure 6. Note that for the 64-bit binaries F and H there are a number of `correct` responses, which, technically, means that the injected fault is not detected. However, the `correct` responses occur when faults affect load operations that are part of a call to the memory library function. Although this function call is present in the IR, it is not protected by the AN encoding scheme from Section 2.3 since no data is actually loaded into the program. The response `correct` ensues since the injected faults affect only those portions of the copied data that are subsequently not used and hence not loaded into the program.

It is interesting to compare the total numbers of fault injection experiments with those for AN encoding only. Figure 6a, for the 32-bit binaries, is dominated by `backend` responses. Moreover, the total numbers of fault injection experiments in Figure 6a are nearly
twice as high as in Figure 5a since the modified backend from Section 3 duplicates the vast majority of load operations. For the 64-bit binaries, on the other hand, backend responses do not dominate Figure 6b as clearly. This is, once again, in agreement with the fact that there is lower register pressure on the x86_64 architecture, causing the backend to insert fewer additional memory accesses.

5.5 Runtime overheads

Fault tolerance measures come at the price of performance penalties since some form of redundancy is required. The runtimes for the test programs from Table 1 are depicted in Figure 7, where geometric means across all test programs are shown. Since runtimes are normalized to the plain binaries, overheads due to AN encoding and the backend modifications can be read off immediately.

Figure 7 shows that the largest fraction of runtime overhead is due to AN encoding. This is plausible since the fundamental operations of encoding and decoding are implemented using expensive integer multiplication and division. AN encoding is known to introduce large overheads [19, 26, 40, 42]. As for the overheads due to backend modifications, the duplication of register spills is the most expensive modification for 32-bit binaries, followed by the duplication of function arguments. This is in agreement with the fact that i386 has relatively few registers and uses a calling convention by which all arguments are passed on the stack. Neither of these observations apply to the x86_64 architecture, and hence the overheads introduced by the backend modifications are considerably lower.

Runtime overheads of the backend modifications have also been evaluated on a subset of the SPEC CINT2006 suite. Figure 8 shows the geometric means of the runtimes of the six benchmark tests listed at the end of Section 4. Again, runtimes are normalized to the plain binaries, to which no backend modifications have been applied. Note that the overhead introduced by duplicating arguments is lower in Figure 8a than in Figure 7a. An explanation for this is that functions in the SPEC benchmarks have larger bodies, and hence longer execution times, than in the test programs from Table 1. Therefore, the overhead introduced by duplicated function arguments carries less weight. When all backend modifications are applied, the resulting mean overhead is 1.50x for 32-bit binaries and 1.13x for 64-bit binaries.

As noted in Section 4, AN encoding may not produce correct programs if the values to be encoded take up too many bits. Therefore, AN encoding has not been applied to the SPEC benchmark tests. For better comparison between the test programs from Table 1 and the SPEC benchmarks, Table 4 lists the overheads for the test programs normalized to the binaries with AN encoding. The numbers roughly reflect the heights of bars in Figure 8. Note that multiplying the overheads listed in Table 4 for the individual backend modifications leads to a larger number than the overhead of all modifications. This is caused by reserving the register ebx (rbx respectively), which is required for the modifications return, arg, jt, spill. The overhead due to reserving this register appears for each of these modifications. Thus, when multiplying these overheads, the reserved register is accounted for four times, while the combination of all modifications only pays for this once. Note also that the penalty for reserving registers ebx and rbx respectively is very low, as evidenced by the runtime overheads of the jt and return modifications. This is particularly remarkable given that i386 has only eight general purpose registers.

All runtime measurements were conducted on an Intel Core i7-4790 CPU running at 3.6GHz. Total system memory is 32GB. The operating system is Ubuntu 16.04.1 LTS, with a 4.4.0 Linux kernel.

6. Related work

Fault tolerance schemes that are applied by program source transformation appeared early [38]. Although such schemes have very limited control over the code generation process in the compiler, low rates of silent data corruption can be achieved. However, a considerable proportion of faults still lead to program crashes [28]. With the advent of super-scalar processors it became viable to implement DMR-based error detection schemes by duplicating machine instructions [37]. Subsequently proposed fault tolerance schemes were also implemented by modifying compiler backends, e.g. [14, 33, 39, 51]. Unlike in this report, these schemes usually
The return address can be thought of as a control-flow protection mechanism. The presented backend modification that detects errors in the signatures is under a point to an object’s virtual function table must be protected against memory faults [7]. This applies to object-oriented languages, such as C++, and thus goes beyond the scope of this report, where the sole focus was on a C language backend. The fault tolerance schemes in [7, 8] were implemented based on aspects [47]. Conceptually, aspects operate on program source code, but their implementation requires interaction with the compiler, over which the user has no detailed control. This, again, opens up the possibility that the implementation of aspects introduces new vulnerabilities.

Recently, an aspect-based implementation of AN encoding has appeared [26]. AN encoding was originally introduced in [10] and studied in detail, among other arithmetic error codes, by [2, 21]. Protecting processors by AN encoding was suggested in [20], where the ANB and ANBD schemes were also introduced. IR-based implementations of AN encoding appeared in [19, 40]. Other fault tolerance schemes combine encoding with DMR [11, 28, 35], as was done in this report. However, a key motivation for this report was that, when DMR is applied to memory operations selectively, i.e. only to local memory accesses inserted by the compiler backend, duplication is safe also for multi-threaded programs.

### 7. Summary and Discussion

Fault tolerance schemes that are applied to programs at the level of intermediate representation (IR) cannot address vulnerabilities resulting from later stages of the code generation process. Specifically, it has been shown that an error detection scheme that operates on LLVM IR fails to protect significant numbers of memory accesses against faults. This is because the compiler backend may introduce additional, unprotected memory accesses to implement, e.g., register spilling or function argument passing. This report has presented backend modifications that add error detection to previously unprotected memory accesses by dual modular redundancy (DMR). These modifications, in conjunction with an IR-based scheme, succeed at detecting all errors resulting from single bit flips in memory.

Implementing fault tolerance schemes at the level of IR, or at even higher abstraction levels, ensures target-independence and enhances productivity. The latter is particularly important for relaxed fault tolerance schemes, where some amount of vulnerability is accepted in exchange for reduced overhead [17, 28, 40]. In quantifying the vulnerability level of a relaxed scheme, meaningful results can only be obtained if one is guaranteed that the code generation process following the application of the fault tolerance scheme does not introduce new vulnerabilities. The backend modifications presented in this report give this guarantee, and they can be coupled with arbitrary IR-based schemes.

Approaches to error detection based on DMR can detect all faults that lead to single bit flips. If more than one bit is corrupted by a fault and if the redundant copies of a data word are affected by the corruption in the same way, this cannot be detected. The probability that multiple bit flips result in an undetectable error may be low, and it certainly depends on how far apart in memory redundant data words are stored. This suggests that there may be an interesting trade-off between data locality, which is beneficial for cache performance, and vulnerability. Triple modular redundancy (TMR) enables recovery from single bit flips by majority voting.
cf. [11, 22], but can also guarantee that all double bit flips are detected.

Runtime measurements have shown that the overhead due to the presented backend modifications is, on average, 1.50x for binaries from SPEC CINT2006 running on i386, and 1.13x for the corresponding binaries running on s68K. This is in agreement with the naive expectation that there is less need to protect against faults in the memory system on machines with more registers. The reported runtime overheads are noticeably lower than for the nZDC scheme, which also duplicates memory accesses [14]. This is unsurprising since, in this report, error detection is applied to memory accesses more selectively.

Some of the presented backend modifications required that the register ebx (rbx respectively) be reserved for temporary storage of the flags register. Although the resulting runtime overhead is very low, eliminating the need to reserve registers would be an interesting project from a software engineering point of view since this would likely require tighter coupling of the presented implementation with register allocation. To achieve this in a clean and flexible way, one may want to consider using aspects [47]. Another interesting extension of the work presented in this report would be to add error detection measures to memory accesses inserted by compiler backends for the C++ programming language. It is known that, due to object-oriented concepts, additional memory accesses are required [7], giving rise to new vulnerabilities.

Acknowledgments
This work was funded by the German Research Council (DFG) through the Cluster of Excellence ‘Center for Advancing Electronics Dresden’ (Caed). The authors acknowledge useful discussions with Sven Karol and Tobias Stumpf. The authors would also like to thank Sven Karol for comments on draft versions of this report and Julia Bolotina for proof-reading and editing.

References


