flexMEDiC: flexible Memory Error Detection by Combined data encoding and duplication

Norman A. Rink and Jeronimo Castrillon
Center for Advancing Electronics Dresden
Technische Universität Dresden, Germany
firstname.lastname@tu-dresden.de

***Abstract*—Errors in memory are known to be a major cause of system failures. Moreover, it has recently been found that *single-error correcting, double-error detecting* (SECDED) codes, which are widely used in ECC memory modules, are incapable of handling large fractions of errors that occur in practice. This calls for more powerful error detection measures. However, the higher the number of bit flips that can still be detected as an error, the larger the memory overhead. Cost considerations and the varying needs for reliability of different applications may not always warrant laying down extra hardware to accommodate overheads. Software-implemented error detection offers a flexible alternative. In this work we propose the software-implemented flexMEDiC scheme for detecting errors in the memory system, including main memory, on-chip caches, and load-store queues. It is shown that single and double bit flips are detected by flexMEDiC, and evidence is given that suggests that up to five bit flips within a single data word can still be detected as errors. The average runtime overhead incurred by flexMEDiC is 1.55x.**

I. Introduction

Faults in hardware are increasingly becoming a problem for the reliability of computing applications. Faults lead to erroneous application behavior with non-negligible probabilities [1], [2], [3], and can cause entire system outages [4]. Growing fault rates are a consequence of the continuing trend toward smaller feature sizes and tighter integration [5], [6], [7], [8]. Moreover, variability in manufacturing processes and measures to limit the temperature and energy footprint of devices further reduce reliability [9], [10], [11].

Studies have shown that errors in DRAM are a major cause of system failures and service disruptions [12], [1], [2]. In the future, the vulnerability of systems to memory errors is likely to increase as a result of the current trend toward reducing power consumption and utilizing dim and dark silicon. Specifically, to save energy, it has been suggested that refresh cycles of DRAM modules be extended [13], [14], and that operating voltages of SRAM be lowered [15]. Both strategies negatively affect the capability to retain data and hence will lead to errors in memory. When SRAM modules become unreliable, memory errors are no longer only a problem for main memory, but will also affect on-chip memories such as caches and load-store queues. The need to protect load-store queues has recently been stressed [16].

Services and applications with strict availability and reliability requirements can be protected against memory errors by employing ECC memory, which commonly relies on *single-error correcting, double-error detecting* (SECDED) codes. However, simple SECDED codes have been found insufficient to protect against significant portions of memory errors [17]. Employing more complex measures in hardware may be ruled out by cost considerations. In particular, on-chip memories may not be able to afford the additional area required to implement more complex codes for error detection.

Alternatively, error detection and correction, typically subsumed as *fault tolerance*, can be implemented in software [18]. This has the advantage that costs and overheads can flexibly be adjusted to an application's need for reliability, cf. [19]. Moreover, software-implemented fault tolerance schemes can be retrofitted onto existing systems. Many popular software-implemented fault tolerance schemes rely on *dual modular redundancy* (DMR) to detect errors [20], [21], [22], [16]. Since DMR implies that data is duplicated, a 100% memory overhead would be incurred, which is usually avoided by assuming that ECC memory is used. Given the observed shortcomings of ECC memory [17], there is a clear need for more comprehensive fault tolerance measures.

This work presents flexMEDiC, a software-implemented error detection scheme that can be configured to detect varying numbers of bit flips in memory at adjustable overheads. The flexMEDiC scheme achieves this by encoding data with the simple, yet effective, *AN encoding* scheme [23], [24]. The code words are precisely the multiples of a fixed integer constant A. Errors are detected when non-code words are encountered. The choice of A determines the minimum distance between code words and hence the maximum number of flipped bits that can still be detected as an error. The memory overhead also depends on the constant A. Thus, by varying A, memory overhead and error detection capability can flexibly be traded for each other.

Generally, AN encoding and related schemes [25], [26] can be applied to facilitate *encoded processing* [27], which serves to detect errors in both memory and the CPU. Encoded processing necessitates that standard arithmetic and logical operations be replaced with their encoded counterparts, leading to large runtime overheads of up to several 10x or 100x [28]. The purpose of flexMEDiC is to protect the memory system, including load-store queues and on-chip caches, but not including the CPU's register file. This implies that only load and store operations must be replaced with their encoded variants. As a result, flexMEDiC incurs an average runtime overhead of 1.55x, significantly lower than encoded processing. It is shown that all single and double bit flips in memory are detected by flexMEDiC. We also present evidence that up to five bit flips within a data word can be detected when $A = 58659$ [29], with a memory overhead of 16 bits per data word.

For maximum flexibility and extensibility, flexMEDiC implements AN encoding at the level of LLVM intermediate representation (IR) [30]. It is shown that this leaves a number of memory accesses vulnerable to errors, namely those accesses that are inserted by the compiler backend when IR is lowered to machine instructions. Therefore, the second component of flexMEDiC, next to AN encoding, is a backend that applies DMR-based error detection to the memory accesses it inserts.

This work is structured as follows. Section II introduces AN encoding and identifies its shortcomings when applied at the IR level. Section III explains the structure and implementation details of flexMEDiC. Section IV introduces the suite of test programs on which flexMEDiC is evaluated in Section V. Section VI discusses related work, and Section VII summarizes the findings of the present work.

II. Background

The flexMEDiC scheme facilitates the detection of errors in memory by encoding data words before they are stored to memory. Thus, in the absence of errors, only valid code words reside in memory, and whenever a non-code word is loaded from memory, an error is detected. In the flexMEDiC scheme, integer data is encoded by multiplying with a fixed integer constant A. Hence, the valid code words are precisely the multiples of A. This is known as *AN encoding* [23], [24].

A. AN encoding

To achieve that only valid code words reside in memory, integer values must be *encoded* before being stored, i.e.

$$m_{encoded} = m \cdot A, \tag{1}$$

for any integer m. Consequently, whenever a value $m_{encoded}$ is loaded from memory, it must be *decoded* before further processing takes place, i.e.

$$m = m_{encoded}/A. \tag{2}$$

Thus, errors can be detected by evaluating the following boolean expression for a value n that has been loaded from memory:

$$n \bmod A = 0. \tag{3}$$

In the absence of errors, the value n is a valid code word, and hence expression (3) evaluates to `TRUE`. If expression (3) evaluates to `FALSE`, an error must have occurred in memory.

Like all error detection schemes, AN encoding relies on redundant information. Specifically, if the value m is represented by k_m bits, and A is represented by k_A bits, then the encoded value $m_{encoded}$ requires $k_m + k_A$ bits. This means that k_A bits are used to store redundant information, and that the memory overhead is k_A bits per data word.

The maximum number of flipped bits that can still be detected as errors is one less than the minimum Hamming distance between code words. Previous work has studied how this distance varies with the choice of A [29], [31]. Generally, the larger the minimum Hamming distance, the larger k_A and hence the memory overhead of AN encoding. In flexMEDiC, the choice of the constant A is left to the user, who can thus flexibly trade memory overhead for the maximum number of detectable bit flips.

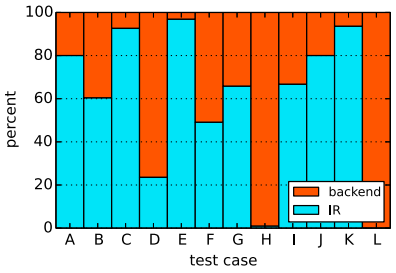


Fig. 1: Dynamic load operations present in intermediate representation (IR) or inserted by the compiler backend.

Implementing AN encoding for data in memory requires that load and store instructions be instrumented with operations that facilitate encoding and decoding, i.e. multiplication and division respectively, cf. (1) and (2). This is conveniently done by instrumenting programs at the level of LLVM intermediate representation (IR) [30], which makes flexMEDiC readily extensible. However, in lowering the IR to machine instructions, the compiler backend may insert additional memory accesses to handle, e.g., register spills. At the IR level, such accesses cannot be protected against errors.

B. Additional vulnerabilities

Figure 1 shows in blue the percentages of dynamically executed load operations that originate from load instructions present in the IR of twelve test programs. Load operations that are inserted by the backend are displayed in red. The test programs are labeled A–L (cf. Table I in Section IV). Evidently 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.

Compiler backends for the C programming language insert additional memory accesses for the following purposes: to handle register spills; to save and restore callee-saved registers, the frame pointer, and the return address; to pass function arguments; to access jump tables. When error detection measures are added to programs at the IR or source level, backend-inserted memory accesses will be left vulnerable to errors. To detect errors in these accesses, the flexMEDiC scheme's second component, next to AN encoding, is a modified compiler backend that uses DMR to protect all additional memory accesses that it inserts.

DMR-based error detection is used for the backend-inserted memory accesses for two reasons. First, the majority of these accesses are required to implement function calls. By using DMR instead of AN encoding, the high-latency integer operations from (1)–(3) can be avoided, thus keeping function calls efficient. Second, flexMEDiC is capable of detecting multiple bit flips at the IR level, and this capability should not be undermined by the backend, which necessitates that memory accesses be fully duplicated.

All of the backend-inserted memory accesses, apart from accesses of jump tables, operate on the local program stack. Therefore, they can safely be duplicated also in multi-threaded

Listing 1: Store and load instructions with encoding and decoding.

```
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 i1 %4, label %next_bb,
                label %exit_bb

next_bb:
      %5 = sdiv i64 %2, %A     ; decode
      ...

exit_bb:
      call void @exit(i32 ENCODING)
```

Listing 2: Load instruction with optimized checking.

```
some_bb:
      ...
      %1 = mul i64 %0, %A      ; encode
      store i64 %1, i64* %p
      ...
      %2 = load i64* %p
      %5 = sdiv i64 %2, %A     ; decode
      %3 = mul  i64 %5, %A
      %4 = icmp eq i64 %3, %2  ; check
      br i1 %4, label %next_bb,
                label %exit_bb

next_bb:
      ...

exit_bb:
      call void @exit(i32 ENCODING)
```

applications. This is also true of the jump table accesses as they are read-only. No memory accesses are duplicated for AN encoding. Hence, flexMEDiC can immediately be applied to multi-threaded applications.

III. Implementation

The flexMEDiC scheme has been designed with maximum flexibility and modularity in mind. Its implementation consists of two major components: the *AN encoder* and the *DMR-enhanced backend*, both of which can be configured and used independently, and can also be freely combined with other fault tolerance schemes. We have implemented the AN encoder as a compiler pass that operates on LLVM IR, and the DMR-enhanced backend has been obtained by modifying the LLVM backend for the *x86* architecture. Figure 2 depicts how flexMEDiC generates executable binaries that are protected against memory errors.

A. The AN encoder

Listing 1 demonstrates how load and store operations are instrumented at the IR level to implement the AN encoding scheme that was introduced in Section II-A. Error checking is performed immediately after load instructions. If a check fails, the program exits with the special exit code `ENCODING`, indicating that an error has been detected.

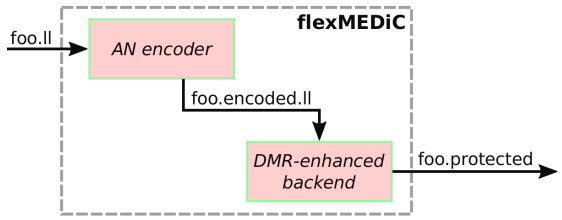


Fig. 2: Generation of protected binaries with flexMEDiC.

It has already been noted that the integer operations that are used to implement the AN encoding scheme have high latencies. Checking and decoding can be optimized if decoding is performed first, cf. [32]. Then, the decoded value can be used to check for errors with a multiplication rather than a modulo operation, as in Listing 2. This is beneficial for performance since multiplication typically has lower latency than modulo. Note that optimizers cannot remove the sequence of `sdiv` and `mul` instructions in Listing 2 since, when operating on integers, division and multiplication are not inverse operations.

The AN encoder is configurable as the constant A can be chosen freely at application compile time. It is flexible since, as an LLVM IR pass, it can easily be modified and extended.

B. The DMR-enhanced backend

The backend-inserted memory accesses that must be protected against errors have been listed in Section II-B. The DMR scheme we have implemented operates as follows. Whenever a value is stored to memory, typically from a register, a second copy of the value is written to an independent location. When the value is re-loaded, the two copies are compared. Disagreement between the two copies indicates the presence of an error in memory. After an error has been detected in a backend-inserted memory access, the executing program is terminated with the special exit code `BACKEND`.

The remainder of this section discusses how the individual backend-inserted memory accesses are handled in flexMEDiC. Please refer to the accompanying technical report [33] for full implementation details. While the wide-spread *x86* architecture is used to demonstrate the functionality of flexMEDiC, compiler backends that target other processor architectures can use analogous DMR-based measures to protect the memory accesses they insert.

1) Register spills (spill): Typical instruction sequences for spilling and restoring a register are shown in Listing 3. Values are spilled to the function's stack area, which is addressed

Listing 3: Spill and restore.

```
mov   rcx,-0x30(rbp)
  ...
mov   -0x30(rbp),rcx
add   rcx,rsi
```

Listing 4: DMR-protected spill and restore.

```
mov   rcx,-0x38(rbp)
mov   rcx,-0x30(rbp)
  ...
mov   -0x30(rbp),rcx
cmp   -0x38(rbp),rcx
jne   <exit>
add   rcx,rsi
```

relative to the frame pointer in the `rbp` register. Additional instructions are needed to duplicate the value of the spilled register on the stack, cf. Listing 4. Error detection is facilitated by the `cmp` and `jne` instructions in Listing 4: when the comparison fails, control is transfered to an `exit` block which terminates the program with the exit code `BACKEND`.

2) Return address (return): On the *x86* architecture, the return address is passed on the stack. Hence, given the possibility of memory errors, the return address can never be assumed to be correct. We have modified the function calling convention so that the return address is passed in register `rbx`, which we reserve for this purpose. The first instruction in the called function pushes `rbx` onto the stack, as in Listing 5. Now the two copies of the return address reside side-by-side on the stack. When the called function returns, one copy of the return address is popped off the stack into register `rbx`. To check for errors, the remaining copy on the stack is then compared with the value in `rbx`. To return from the function call, the *x86* `ret` instruction cannot be used since this entails reading the return address from the stack, which constitutes a vulnerability to memory errors. Instead, an indirect jump to the checked return address is performed, cf. Listing 5, which necessitates that the stack pointer (`rsp`) is incremented manually.

Listing 5: DMR-protected return address.

```
push   rbx
  ...
pop    rbx
cmp    (rsp),rbx
jne    <exit>
add    0x8,rsp
jmp    *rbx
```

Note that on architectures with a designated return register, e.g. ARM or MIPS, protecting the return address against memory faults does not require that an additional register be reserved or that the calling convention be modified.

3) Function arguments (arg): When function arguments are passed on the stack, they are vulnerable to memory errors. To enable error detection, the calling convention has been modified so that a duplicated copy of the argument sequence is also put on the stack, next to the original sequence, cf. Figure 3. Whenever one of the original arguments is loaded into a register, it is compared with the corresponding argument in the duplicated argument sequence.

Note that the standard calling convention on *x86* has been modified in two ways. First, the return address is passed in the

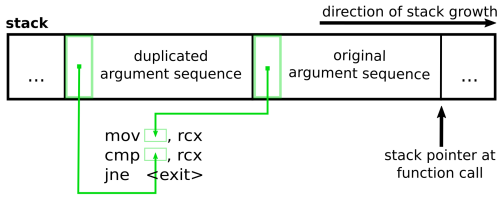


Fig. 3: DMR-protected function arguments on the stack.

register `rbx`, in addition to being put on the stack by the `call` instruction. Second, a duplicated argument sequence resides on the stack immediately above the original arguments. The obligation to implement this calling convention rests entirely with the caller. Thus, if a callee does not perform error checking for the return address or arguments, the function call will still work. In particular, library functions can 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 `rbx` register nor the stack will be set up according to our modified calling convention. Hence, executing the unprotected function will lead to premature program termination with exit code `BACKEND`.

4) Callee-saved registers (csr), frame pointer (fptr): Callee-saved registers and the frame pointer are pushed onto the stack at function entry, and restored immediately before returning. Therefore, protecting these stack accesses is completely analogous to the handling of the return address.

5) Jump tables (jt): Jump tables are an efficient way of implementing switch statements. A jump table is an array of addresses of basic blocks. Unlike the previously discussed memory accesses, jump tables do not reside on the stack, but in the code segment. To protect jump tables against errors, the backend duplicates each jump table. Before jumping to a jump table entry, the entry is loaded into a register and compared with the corresponding entry in the duplicated jump table.

The full flexMEDiC scheme requires that all backend-inserted memory accesses are equipped with error detection measures. However, the individual DMR-based measures that have been presented in this section, i.e. *spill*, *return* etc., can be switched on and off independently by compiler flags. This means that flexMEDiC's protection of backend-inserted memory accesses can easily be replaced with alternative implementations if need be.

IV. Test Programs and Code Generation

To demonstrate the effectiveness of flexMEDiC, faults are injected into the test programs in Table I. Due to simple combinatorics, the spaces of faults that can affect a program are quite large. Therefore, conducting exhaustive fault injection is a processor-bound task, which, for the relatively small programs in Table I, can still be carried out in reasonable time.

Some of the test programs (C, E, K) are borrowed from the MiBench suite [34], and similar programs are often used to evaluate fault tolerance schemes, e.g. [20], [35], [16]. The programs represent typical algorithmic tasks, e.g. sorting,

test	*plain*			*encoded*		*fptr*		*csr*		*jt*		*return*		*arg*		*spill*		*all*	
	instr.	ld.	IR	instr.	ld.	instr.	ld.	instr.	ld.	instr.	ld.	instr.	ld.	instr.	ld.	instr.	ld.	instr.	ld.
A	34	10	8	100	10	106	11	100	10	100	10	107	11	100	10	100	10	109	12
B	432	58	35	624	39	630	40	632	40	624	39	631	40	624	39	624	39	641	42
C	112	27	25	268	29	274	30	276	30	268	29	275	30	268	29	268	29	285	32
D	1816	136	32	2056	136	2232	176	2104	144	2056	136	2288	176	2056	136	2056	136	2480	224
E	1926	500	484	5312	508	5336	512	5360	512	5312	508	5340	512	5312	508	5312	508	5396	520
F	454	157	77	1018	171	1046	185	1102	185	1090	195	1073	185	1018	171	1018	171	1313	237
G	597	146	96	1279	146	1399	165	1279	146	1315	158	1418	165	1279	146	1279	146	1492	196
H	813	224	2	1343	226	1455	251	1497	250	1343	226	1523	251	1343	226	1343	226	1765	300
I	198	54	36	451	43	453	44	461	44	450	46	455	47	450	46	463	50	482	53
J	43	10	8	98	10	104	11	98	10	98	10	105	11	98	10	98	10	107	12
K	413	93	87	1104	136	1136	144	1200	144	1104	135	1151	143	1104	135	1247	177	1406	201
L	15	5	0	15	5	19	6	15	5	20	7	20	6	19	7	15	5	37	12

TABLE II: Dynamic instructions and load operations for the test programs on *x86_64*.

graph traversal, manipulation of bit patterns, and linear algebra. Test program L consists of a switch statement that selects one of the many arguments of the enclosing function. It is included here since it is the only program that passes function arguments on the stack in the *x86_64* calling convention.

The test programs are evaluated on the *x86_64* architecture, the 64-bit version of *x86*. Properties of the binaries generated by flexMEDiC are summarized in Table II, namely: dynamically executed instructions (instr.), dynamically executed load operations (ld.), and the number of load operations that are present at the level of LLVM IR (IR). The *plain* block in Table II refers to the binaries without any error detection measures. The *encoded* block refers to the binaries when only AN encoding is applied. Subsequent blocks correspond to the combination of AN encoding with the individual DMR measures in the backend. Finally, *all* refers to the full flexMEDiC scheme, i.e. AN encoding plus *all* backend DMR measures.

All binaries are generated at optimization level `-O3`, and AN encoding is applied with $A = 58659$, cf. [29]. The load operations that are present in the IR are the same for each block of Table II. Therefore, the numbers of load operations are not repeated in blocks other than *plain*. Note that Figure 1 is based on the *plain* block of Table II.

V. 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 flexMEDiC scheme is evaluated by symptom-based fault injection [27], [36], [26]. 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 I, the detailed procedure for which is described in the next section.

	description
A	array reduction
B	bubblesort
C	cyclic redundancy checker (CRC-32)
D	DES encryption algorithm
E	Dijkstra's algorithm
F	arithmetic expression interpreter recursive expression tree evaluation
G	token lexer for arithmetic expressions
H	arithmetic expression parser
I	matrix multiplication
J	array copy
K	quicksort
L	switch

TABLE I: Suite of test programs.

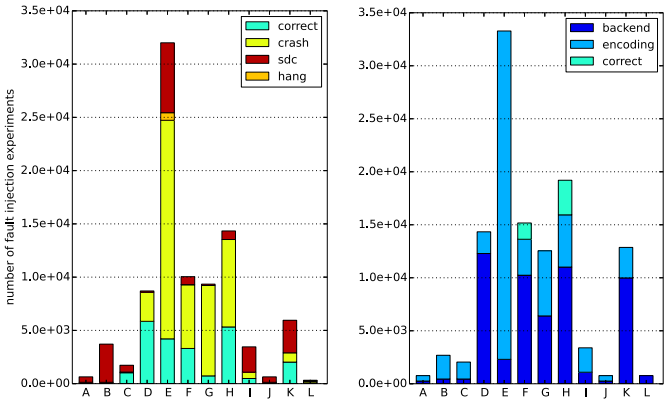
A. Fault injection experiments

The symptom of a fault in the memory system is modeled by flipping a number of bits in the result of a load operation. To inject this symptom into an executing program, the Intel Pin tool [37] 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. E.g., for single bit flips on a 64-bit machine, the number of all possible symptoms is 64 times the number of dynamic 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:

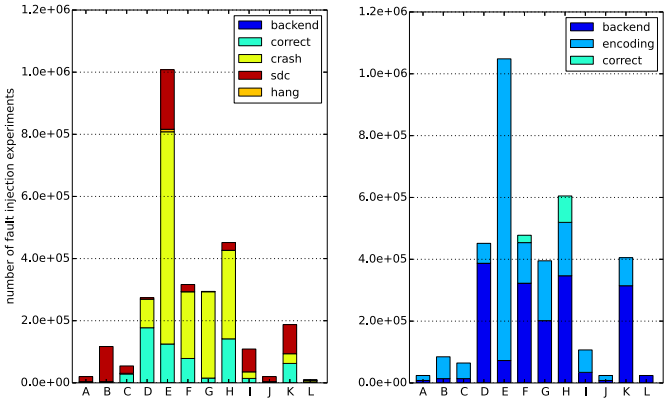
1) *correct*: Despite the fault, the program terminates normally and produces correct output.
2) *hang*: If the program runs for longer than 10x its normal execution time, it is deemed to hang and hence is terminated. In practice, e.g. in safety-critical embedded applications, a hardware watchdog may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

To demonstrate that flexMEDiC can detect multiple bit flips within a single data word in memory, we perform fault injection experiments for the following symptoms: single bit flips, double bit flips, and 5-bit flips. The chosen encoding constant $A = 58659$ is expected to be capable of detecting up to five bit flips [29]. For the single and double bit flips we



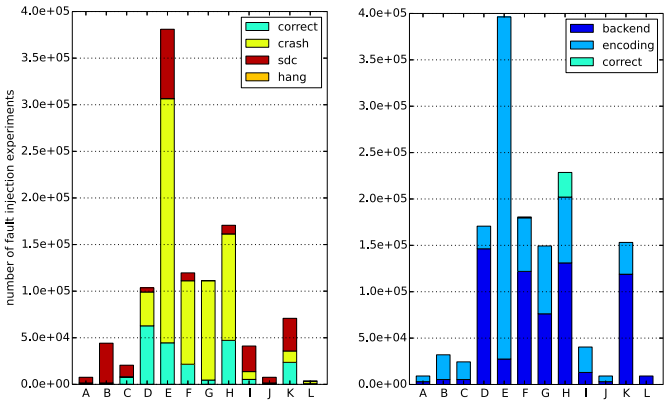
(a) no error detection (b) flexMEDiC

Fig. 4: Single bit flips in memory.



(a) no error detection (b) flexMEDiC

Fig. 5: Double bit flips within a single data word.



(a) no error detection (b) flexMEDiC

Fig. 6: Five bit flips within a single data word; sampling of 0.01% of error patterns.

explore the full space of all possible symptoms. For 5-bit flips this is not feasible in reasonable time since there are almost 4,000 times as many error patterns as for double bit flips. Therefore, we randomly sample 0.01% of all error patterns.

B. Error detection results

Figures 4–6 summarize the results of our fault injection experiments. Note that absolute numbers of fault injection experiments are drawn along the vertical axes. When no error detection mechanisms are used, i.e. for the *plain* binaries, there are noticeable numbers of *crash* and *sdc* responses. On the other hand, when flexMEDiC is applied, all errors are detected as evidenced by Figures 4b–6b. However, for test programs F and H there are a number of *correct* responses, which, technically, means that the injected fault is not detected. The *correct* responses occur when errors affect load operations that are part of a call to the `memcpy` library function. Although this function call is present in the IR, it is not protected by the AN encoder from Section III-A since no data is actually loaded into the program. The response *correct* ensues when errors affect only those portions of the copied data that are subsequently not used.

It is interesting to note that Figures 4–6 show very similar distributions of program responses. This suggests that programs are not particularly sensitive to the number of bits that are flipped; it seems to be more important which load operations, viz. data words, are affected by errors.

C. Runtime overheads

Fault tolerance measures require some form of redundancy, leading to performance penalties. The normalized runtimes for the test programs are depicted in Figure 7, where geometric means across the suite of test programs are shown. The bar labeled *all* depicts the overhead of the full flexMEDiC scheme, which is 1.55x.

According to Figure 7 the largest fraction of runtime overhead is due to AN encoding, which is known to produce large overheads [28], [25], [38]. Table III lists the overheads of the various error detection measures introduced by the DMR-enhanced backend. The numbers in Table III are, again, geometric means across the suite of test programs, and have been normalized to the binaries with AN encoding, i.e. the *encoded* binaries. The total overhead introduced by the DMR-enhanced backend is only 1.07x.

All runtime measurements have been 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.

VI. Related work

Many software-implemented fault tolerance schemes have been proposed [18]. The EDDI [20] and SWIFT [21] schemes detect errors by duplicating a program's data flow, which is achieved by duplicating values in registers and machine instructions. This form of DMR has been developed in various directions. ESoftCheck [39] and Shoestring [19] aim to reduce the overhead of DMR by checking for errors less frequently. DRIFT [40] performs delayed checking to generate faster instruction schedules. The DAFT [22] scheme offloads error detection to separate threads. Modern CPU's SIMD extensions have been used for DMR-based error detection [41] and also for error recovery [42]. All of these schemes, starting with SWIFT, have in common that memory is assumed to be protected by other measures, typically ECC memory.

The nZDC scheme [16] also makes the assumption that memory is already protected against errors. However, instructions that access memory are still duplicated in nZDC since the processor's load-store queue is assumed to be vulnerable to faults. Since the nZDC scheme duplicates all memory accesses, and not just those that access local memory, it is limited in handling multi-threaded applications correctly, as already noted in [16]. The DMR-enhanced backend presented in this work also leads to a significantly lower runtime overhead than nZDC, which is because it is more selective in duplicating memory accesses than the nZDC scheme.

Software-implemented encoded processing of data [27] can detect errors in both the CPU and memory, without the need for hardware support. Protecting processors by AN encoding was suggested in [24], and IR-based implementations appeared in [28], [25], [38]. AN encoding leads to large runtime overheads, which are affected by the scheduling of error checking instructions [32]. Overheads can be reduced when AN encoding is combined with DMR [43], [44], [26]. The present work combines AN encoding and DMR differently from previous schemes in that it applies them at different levels, i.e. at the IR level and in the compiler backend.

That return addresses and frame pointers need protection has already been observed in the context of protecting an operating system against hardware faults [45]. The DMR-based protection of return addresses presented in the present work can be regarded as a mechanism that detects errors in control-flow. General mechanisms for protecting against control-flow errors have also been proposed [35], [46].

VII. Summary and Outlook

Memory errors reduce the reliability of applications. While ECC memory can alleviate the impact of memory errors, it has been found that simple SECDED codes, which are widely used, cannot handle large fractions of errors that affect applications in practice. Software-implemented fault tolerance schemes offer the flexibility of being adjustable to an application's reliability needs. This work has presented the software-implemented flexMEDiC scheme for detecting multiple bit flips in memory. It has been shown that flexMEDiC can detect all single and double bit flips within data words in memory. Furthermore, evidence has been produced that up to five bit flips within a single data word can be detected. The average runtime overhead incurred by flexMEDiC is 1.55x across a suite of representative test programs.

The flexMEDiC scheme has been implemented modularly. Memory accesses that are present in the IR of programs are protected by AN encoding. By varying the encoding constant A, one can trade the memory overhead of AN encoding for its capability to detect multiple bit flips. Additional memory accesses may be inserted by the compiler backend when the IR is lowered to machine instructions. Therefore, flexMEDiC comes with a compiler backend that protects the memory accesses it inserts by DMR. The backend can be configured to protect only subsets of memory accesses, which means that parts of flexMEDiC can easily be replaced with other error detection measures.

We have evaluated flexMEDiC with $A = 58659$, which is known to have good properties [29]. Evidence suggests that up to five bit flips can still be detected as errors with this A. It is generally difficult to determine the minimum distance between code words in AN encoding. Nonetheless, it would be useful to extend flexMEDiC with a mechanism that can automatically find an optimal encoding constant A given the number of bit flips that should still be detectable as errors.

It is worth stressing that, thanks to the AN encoder component, flexMEDiC relies on DMR only for local memory accesses. Hence, no accesses of memory that is potentially shared are duplicated, which implies that flexMEDiC can safely be applied to multi-threaded applications. Future work should verify this.

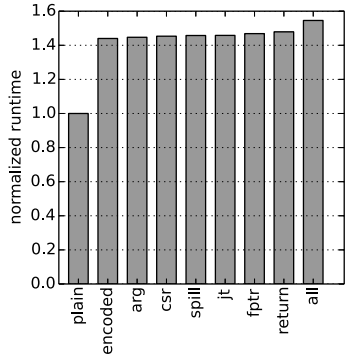


Fig. 7: Mean runtime overheads for the binaries from Table II.

	mean overhead
fptr	1.020
csr	1.009
jt	1.013
return	1.027
arg	1.005
spill	1.012
all	1.073

TABLE III: Overheads normalized to the *encoded* binaries.

Acknowledgment

This work was funded by the German Research Council (DFG) through the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed). The authors would like to thank Sven Karol and Tobias Stumpf for useful discussions.

References

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proc. 11th Int'l joint Conf. Measurement and Modeling of Computer Systems*, ser. SIGMETRICS'09, 2009, pp. 193–204.

[2] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *Proc. 6th Conf. on Computer Systems*, ser. EuroSys'11, 2011, pp. 343–356.

[3] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proc. ACM Symp. Cloud Computing*, ser. SOCC'14, 2014.

[4] (2008) Amazon S3 availability event: July 20, 2008. [Online]. Available: http://status.aws.amazon.com/s3-20080720.html

[5] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. Int'l Conf. Dependable Systems and Networks*, ser. DSN'02, 2002, pp. 389–398.

[6] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[7] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[8] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES'06, 2006, pp. 421–431.

[9] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Ann. Int'l Symp. Computer Architecture*, ser. ISCA'11, 2011, pp. 365–376.

[10] M. B. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *Proce. 49th Ann. Design Automation Conf.*, ser. DAC'12, 2012, pp. 1131–1136.

[11] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The EDA challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *Proc. 51st Ann. Design Automation Conf.*, ser. DAC'14, 2014, pp. 1–6.

[12] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proc. Int'l Conf. Dependable Systems and Networks*, ser. DSN '06, 2006, pp. 249–258.

[13] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," in *Proc. 16th Int'l Conf. on Architectural Support for Programming Languages and Operating systems*, ser. ASPLOS'11, 2011, pp. 213–224.

[14] C. Weis, M. Jung, P. Ehses, C. Santos, P. Vivet, S. Goossens, M. Koedam, and N. Wehn, "Retention time measurements and modelling of bit error rates of WIDE I/O DRAM in MPSoCs," in *Proc. Design, Automation & Test in Europe Conf. & Exhibition*, ser. DATE'15, 2015, pp. 495–500.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'12, 2012, pp. 301–312.

[16] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near zero silent data corruption," in *Proc. Design Automation Conf.*, ser. DAC'16, 2016.

[17] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," in *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'12, 2012, pp. 111–122.

[18] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer, 2006.

[19] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'10, 2010, pp. 385–396.

[20] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Int'l Symp. Code Generation and Optimization*, ser. CGO '05, 2005, pp. 243–254.

[22] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: Decoupled acyclic fault tolerance," in *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques*, ser. PACT'10, 2010, pp. 87–98.

[23] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Trans. Electronic Computers*, pp. 333–337, 1960.

[24] P. Forin, "Vital coded microprocessor principles and applications for various transit systems," in *Control, Computers, Communications in Transportation: Selected Papers from the IFAC/IFIP/IFORS Symposium*, 1989, pp. 79–84.

[25] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "ANB- and ANBDmem-encoding: Detecting hardware errors in software," in *Proc. 29th Int'l Conf. Computer Safety, Reliability, and Security*, ser. SAFECOMP'10, 2010, pp. 169–182.

[26] D. Kuvaiskii and C. Fetzer, "Δ-encoding: Practical encoded processing," in *Proc. 45th Ann. Int'l Conf. Dependable Systems and Networks*, ser. DSN'15, 2015.

[27] U. Schiffel, "Hardware error detection using an-codes," Ph.D. dissertation, Technische Universität Dresden, 2011.

[28] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-encoding compiler: Building safety-critical systems with commodity hardware," in *Proc. 28th Int'l Conf. Computer Safety, Reliability, and Security*, ser. SAFECOMP'09, 2009, pp. 283–296.

[29] M. Hoffmann, P. Ulbrich, C. Dietrich, H. Schirmeier, D. Lohmann, and W. Schröder-Preikschat, "A practitioner's guide to software-based soft-error mitigation using AN-codes," in *Proc. 15th Int'l Symp. High-Assurance Systems Engineering*, 2014.

[30] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. Int'l Symp. Code Generation and Optimization*, ser. CGO'04, 2004, p. 75.

[31] M. Werner, T. Kolditz, T. Karnagel, D. Habich, and W. Lehner, "Multi-GPU approximation methods for silent data corruption of AN codes," in *Proc. 12th Int'l Workshop Boolean Problems*, ser. IWSBP'16, 2016.

[32] N. A. Rink and J. Castrillon, "Improving code generation for software-based error detection," in *Proc. 1st Int'l Workshop Resiliency in Embedded Electronic Systems*, ser. REES'15, 2015.

[33] ——, "Comprehensive backend support for local memory fault tolerance," Technische Universität Dresden, Tech. Rep. TUD-FI-16-04, 2016.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int'l Symp. Workload Characterization*, ser. IISWC'01, 2001, pp. 3–14.

[35] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. on Reliability*, vol. 51, no. 2, pp. 111–122, 2002.

[36] D. Behrens, M. Serafini, S. Arnautov, F. P. Junqueira, and C. Fetzer, "Scalable error isolation for distributed systems," in *Proc. 12th USENIX Conf. Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 605–620.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *Proc. Conf. Programming Language Design and Implementation*, ser. PLDI'05, 2005, pp. 190–200.

[38] N. A. Rink, D. Kuvaiskii, J. Castrillon, and C. Fetzer, "Compiling for resilience: The performance gap," in *Proc. Mini-Symp. Energy and Resilience in Parallel Programming*, ser. ERPP'15, 2015.

[39] J. Yu, M. J. Garzarán, and M. Snir, "ESoftCheck: Removal of non-vital checks for fault tolerance," in *Proc. 7th Ann. Int'l Symp. Code Generation and Optimization*, ser. CGO'09, 2009, pp. 35–46.

[40] K. Mitropoulou, V. Porpodas, and M. Cintra, "DRIFT: Decoupled compiler-based instruction-level fault-tolerance," in *Proc. 26th Int'l Workshop Languages and Compilers for Parallel Computing*, ser. LCPC'13, 2014, pp. 217–233.

[41] Z. Chen, A. Nicolau, and A. V. Veidenbaum, "SIMD-based soft error detection," in *Proc. ACM Int'l Conf. Computing Frontiers*, ser. CF'16, 2016, pp. 45–54.

[42] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer, "Elzar: triple modular redundancy unsing Intel AVX," in *Proc. Int'l Conf. Dependable Systems and Networks*, ser. DSN'16, 2016.

[43] N. Oh, S. Mitra, and E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Computers*, vol. 51, no. 2, pp. 180–199, 2002.

[44] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Int'l Conf. Dependable Systems and Networks*, ser. DSN'06, 2006, pp. 83–92.

[45] C. Borchert, H. Schirmeier, and O. Spinczyk, "Return-address protection in C/C++ code by dependability aspects," in *Proc. 2nd Workshop Software-Based Methods for Robust Embedded Systems*, ser. SOBRES'13, 2013.

[46] R. Vermu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," in *Proc. IEEE Int'l Test Conf.*, 2007, p. 1010.