This work has been published in the Proceedings of the eighth Symposium on Microcomputer and Microprocessor Applications, Vol. 1, Budapest Technical University, Hungary, October 12-14, 1994

Fault Injection Based Validation of Fault-Tolerant Multiprocessors

V. Sieh[‡], A. Pataricza[†,‡], B. Sallay[†], W. Hohl[‡], J. Hönig[‡], B. Benyó[†]

[†] Department of Measurement and Instrumentation Engineering
Technical University of Budapest
Müegyetem rkp. 9
H-1521 Budapest, Hungary
email: sallay@mmt.bme.hu

[‡] Department of Computer Science III
University of Erlangen-Nürnberg
Martensstr. 3
91058 Erlangen, Germany
email: vrsieh@informatik.uni-erlangen.de

Introduction

One of the most crucial tasks in the design of fault-tolerant computers is the validation of the built-in error detection and handling mechanisms. Predesign validation techniques, like performability modelling and analysis, often require such information as exact failure rates, which is usually unavailable for the user. Moreover, the majority of computer failures originate from transient faults, occurring one order of magnitude more often than permanent ones. The rate and the site of transient faults depends heavily on the lay-out, i.e. experimental validation on a prototype is always required for a realistic prediction of the fault tolerance measures. Software methods dominate in this field, because of the difficulties in hardware-based fault injection methods.

Descriptions at several abstraction levels are used to provide a basis for simulation and fault injection. In this paper, a novel technique is presented that allows realistic fault modelling and injection-based validation of general purpose microprocessors based on a register transfer level VHDL description. In the first section, examples are given for various levels which have been used previously to perform simulation. The reasons for choosing register transfer level for modelling are described with emphasis on general methods of describing and modelling a general purpose microprocessor. A project carried out on a Motorola 88100 microprocessor will serve as an example. Finally, a comparative evaluation of the measurement results with those of the traditional error level injection experiments will be presented.

Previous and present methods

The lowest level of abstraction in fault modelling is the transistor level. The effects of physical faults, such as charge faults resulting from ion radiation, were analysed by using an analog simulation of the physical lay-out. This approach results in a very realistic, but immense model, even if hybrid modelling is used, and only the fault site is modelled in full details. Similar consequences can be drawn for the gate level where faults are manifested as stuck-at or coupling faults.

At the register transfer level, devices are considered to be registers and sets of functions defined on them. Faults occur here as errors in these functions. Compact and general models can be achieved with high-level hardware description languages like VHDL. Models are portable, because functional descriptions are technology-independent. In addition, the greatest advantage of this abstraction level is that it supports the use of functional fault models validated at a lower level of abstraction.

The vast majority of fault injecting applications are implemented at an error level where the manifestation of physical faults is modelled as random data errors in the registers. Various models can be used: single bit errors, multiple bit errors (according to a p-parameter binary symmetrical channel), or random word errors. Fault injecting programs simply interrupt the main application and change the register state at the moment of fault injection, according to the fault model. Compact and portable models can be obtained without any special knowledge on the hardware structure. However, the validity of this fault model is doubtful, since there is no evidence that physical faults disturb the registers in the way expected in the injection method.

In the following, a more detailed and more realistic approach to fault modelling will be presented.

Functional CPU fault models

The basic fault model used in the fault injector was originally elaborated by Thatte and Abraham in the field of automatic test pattern generation for permanent faults [1]. It was validated for numerous processor types by estimating the fault coverage of the test generation algorithms using this approach on gate-level fault models, showing an excellent coverage of typically higher than 90%.

According to the original model, a general purpose microprocessor is regarded as a set of the following five functions: *data storage, data transfer, data manipulation, register decoding, instruction decoding and execution* (Table 1). Faults in the data storage function, originating from memory cell stuck-at faults and alpha particle radiation, result in bit errors in the registers. Data transfer faults caused by transfer path stuck-at or cross-coupling faults cause bit errors in the destination registers. There is no explicit fault model proposed for the data manipulation function, due to the wide variety of the manipulating operations and the technology-dependency of their implementation. We do not deal with this fault class now, either.

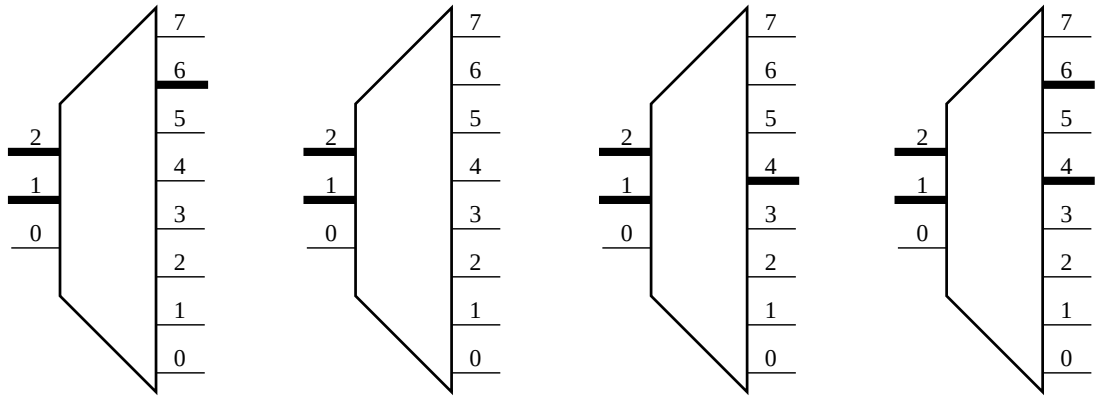
<table>
<tr><th colspan="2">Thatte-Abraham fault model</th><th colspan="2">Brahme-Abraham fault model</th></tr>
<tr><th>Function</th><th>Fault</th><th>Function</th><th>Fault</th></tr>
<tr><td>data storage</td><td>bit errors</td><td colspan="2">same as in Thatte-Abraham</td></tr>
<tr><td>data transfer</td><td>bit errors in
destination registers</td><td colspan="2">same as in Thatte-Abraham</td></tr>
<tr><td>data manipulation</td><td>function dependent</td><td colspan="2">same as in Thatte-Abraham</td></tr>
<tr><td>register decoding</td><td>inactive:
READ - constant
WRITE - no effect
wrong:
READ - wrong source
WRITE - wrong destination
additional:
READ - or/and
WRITE - both written</td><td colspan="2">same as in Thatte-Abraham</td></tr>
<tr><td rowspan="2">instruction decoding
and execution</td><td rowspan="2">inactive:
NOP
wrong:
wrong instruction
additional:
two instructions
simultaneously</td><td>microinstruction</td><td>inactive
wrong
additional</td></tr>
<tr><td>microorder</td><td>inactive
wrong
additional</td></tr>
</table>

Table 1: Functional fault models of a CPU

The functional fault model of the remaining two functions is based on the *general address decoder fault model* (Fig. 1). During normal operation, the appropriate output line is selected. In the faulty case, no or wrong output line is activated, or multiple output lines are asserted selecting either no, wrong, or multiple decoded objects, respectively. For the register decoding, decoded objects mean registers selected for read or write; for the instruction decoding, they denote instructions selected for execution, respectively.

The concept of *microinstructions* and *microorders* was introduced as a refined model by Brahme and Abraham, even for non-microprogrammed control units [2]. An instruction is decomposed into a sequence of consecutively executed microinstructions, each of them is regarded as a set of concurrently executed microorders. In this refined fault model, either microinstructions or microorders can be selected wrongly instead of whole instructions.

It must be noted that decoding faults have a significant probability even in today's technology. The rate of bit changes resulting from alpha particle radiation is proportional to the silicon area of the unit; the size of the instruction decoder and scheduler units is comparable with that of the register array, especially in case of hardwired RISC microprocessors. The instruction execution unit is also sensitive to



a/ Normal operation b/ No selected object c/ Wrong selected object d/ Multiple selected objects

Fig. 1: General fault model of an address decoder

transient power noises and other dynamic failures.

In our approach, the very same fault model was used as described above, but using the single transient fault assumption instead of the permanent one. This assumption even simplifies the construction of realistic fault models, compared with the permanent fault model. In the latter case a faulty unit can be shared among different operations, hence it can induce errors in different functional operations. The correspondence between functional units and operations is needed to create an exact description of the fault effects. However, a description and a fault model in pure behavioural forms are essentially sufficient for the modelling of single transient faults affecting only a single operation.

Fault injector for the MC88100

The rest of the paper will be illustrated with the injector for the Motorola 88100 microprocessor, used in the MEMSY ([6]) multiprocessor system. It is a fully pipelined RISC processor with a Harvard architecture (Fig. 2) assuring an effective speed as high as one instruction per clock cycle [5]. The proc-

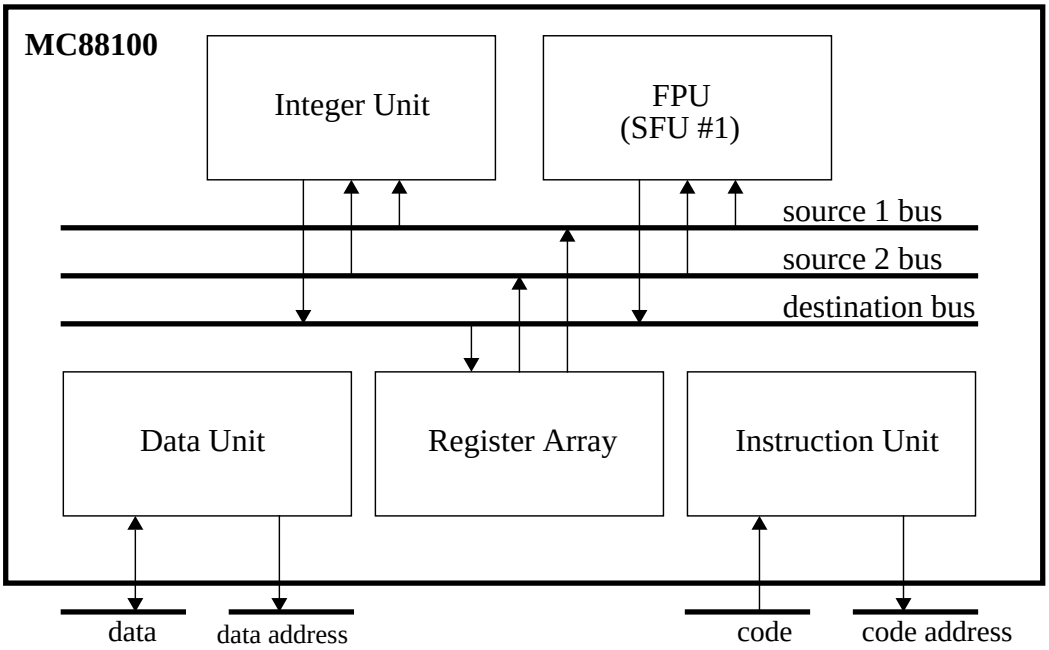


Fig. 2: Functional block scheme of the Motorola MC88100

essor contains three internal buses (two for operands and one for results) and five main functional units. The instruction unit implements the instruction fetch, decoding and execution functions. The data unit is responsible for the data memory accesses. The register file is a general purpose register array. The integer unit performs data manipulation operations in a single clock cycle while the FPU performs float-

ing point operations.

Some restrictions were made in the modelling of the processor, like the confinement to the *user programming model* where the applications under observation run. Thus trap instructions, entering the supervisor mode and control register transfer instructions inaccessible in user mode are omitted. The FPU - declared by Motorola a special function unit not belonging closely to the architecture - was omitted from the model as it does not affect the execution core of the microprocessor. Only the execution sequence of a single instruction was the subject of interest from the point of view of fault injection, therefore neither pipeline constructs nor delayed branching are explicitly described, although their faults are still covered by the functional model.

70-85% of the instruction set of typical benchmark programs, e.g. *dhrystone, whetstone, multigrid*, are still involved in the model. This ratio is well over 90% for regular programs like *ls, ps*, etc.

Modelling technology

The MC88100 was described using a VHDL subset especially developed for describing microprocessors. VHDL [4] has significant advantages: it has become today's de facto standard hardware description language that supports hierarchical and mixed models at a wide range of abstraction levels, from the gate level up to the behavioural levels, and advanced VHDL simulator tools can be used for the validation of the processor model.

In general, VHDL has two basic concepts: structural and behavioural views [3]. The main emphasis in a structural description is on the components of the entity, and on their interconnections. The behavioural point of view focuses on the function performed by the entity, regardless of its actual physical lay-out. In VHDL, these two aspects can be mixed, allowing the user to describe the part in focus precisely and the parts of secondary importance less accurately.

We found that model compactness and transformability can be achieved by the definition of the processor registers and state variables as objects,[1] and by a behavioural description with high level sequential and manipulative statements. However, the identification of the execution sequence and the undocumented subunits remains still a problem. Only the user manual is usually available for the model developer, which typically contains only a rough and verbal specification of the structure and the instruction set, insufficient to create a VHDL equivalent. The processor model must be refined until a satisfactorily clarified structure and the microinstruction - microorder decomposition of every instruction is reached. Although this task appears to be indeterminate, there are many principles we can rely on. Causal operation can be always assumed, by which transfer paths and microinstruction sequences can be well estimated. Exact timing information can also help determine the execution sequence if provided. It can be also assumed that minimal hardware was used to implement the desired functions. Knowledge of the exact mapping between units and operations is not needed, since only transient faults are examined. It can be supposed that instruction codes are compact; that is, if the codes of one instruction class differ only in a short fragment, this fragment is likely to be decoded later, by one functional unit. Applying these assumptions, it is possible to achieve finally a good approximation of the register and the microorder set of which the behaviour of the microprocessor can be formally described.

The microprocessor description style

The compiler developed to translate the VHDL-script into a simulator accepts a sequential subset of VHDL and a special description style conforming to the functional model. It is suitable to describe any current general purpose microprocessors.

Processor interface signals are declared in the entity header of such a description. The architecture declaration part may contain function definitions, which keep the model compact and structured. The whole architecture body consists of a single process statement providing the timing frame for one instruction cycle. State variables and microorders (defined as sequential procedures withour arguments) can be declared in the header of this process. Microorder procedures contain very simple assignment statements.

The decoding and sequencing of an instruction is described in the process body. Every sequential

[1] In VHDL, an object is either a signal, a variable, or a constant.

statement, like variable and signal assignment, procedure call and return, if-case-loop, null statement is implemented. Thus the CPU description may contain conditional control flow statements (instruction decoding) and procedure calls (microorder activation). Exception condition checks should be explicitly built in the VHDL description in the form of assertions, since injected faults may result in exceptions during simulation. The use of types is restricted to the predefined discrete set consisting of {Bit, Bit_Vector, Register_Array, Integer, Boolean} types, characteristic to microprocessors. Certain predefined attributes (’Left, ’Right, ’Range), sliced and indexed names are supported.

Wait statements, that denote the end of microinstructions, group microorders into time frames. Since it is a suitable moment for fault injection, wait statements are compiled into fault injecting function calls.

The following fragment shows typical examples for each construct:

```
entity m88k is port (
  -- I/O signals to be declared here
-- example:
  data_address: out Bit_Vector( 31 downto 0 );
   );
  end m88k;

architecture m88k_behavioural of m88k is
  -- utility functions to be defined here
  -- example:
  function Bit_to_Integer( arg: Bit ) return Integer is
  begin
     if arg = '1' then return 1;
        else return 0;
     end if;
  end Bit_to_Integer;
begin
  process
  -- description of one instruction cycle
  -- declaration of state variables
  -- example:
     variable D_bus: Bit_Vector( 31 downto 0 );
  -- definition of microorder procedures
  -- example:
     procedure WriteBack is
     begin
        reg( Decode( IR( 25 downto 21 ) ) ) := D_bus;
     end WriteBack;
  begin
  -- process body, instruction decoding and scheduling description
  -- typical fragment:
     if IR( 31 downto 26 ) = "110010"          -- bsr instruction
        then
           Save_Return_Address;               -- microorder procedure activation
           wait on CLK;                       -- end of microinstruction
           Set_New_Address;                   -- microorder procedure activation
     end if;
  end process;
end m88k_behavioural;
```

Well-known compiler development tools: *lex* and *yacc* were used for compiler building (Fig. 3).

It should be pointed out that the process of compiler generation is accomplished only once; when porting the methodology to another target microprocessor type, no repetition is needed.

The target code of the compiler (Fig. 3) is a C language file implementing a simulator of the fault-free microprocessor. To this module fault injecting routines are appended. It should be mentioned that a fully automatic addition of these functions is still possible, but in the experimental version they were generated manually.

Fault injection techniques

The most appropriate moments for fault injection (except for the register decoding fault class) arrive after the completion of each microinstruction execution, denoted by wait statements. The techniques of fault injection naturally differ for the individual fault classes:

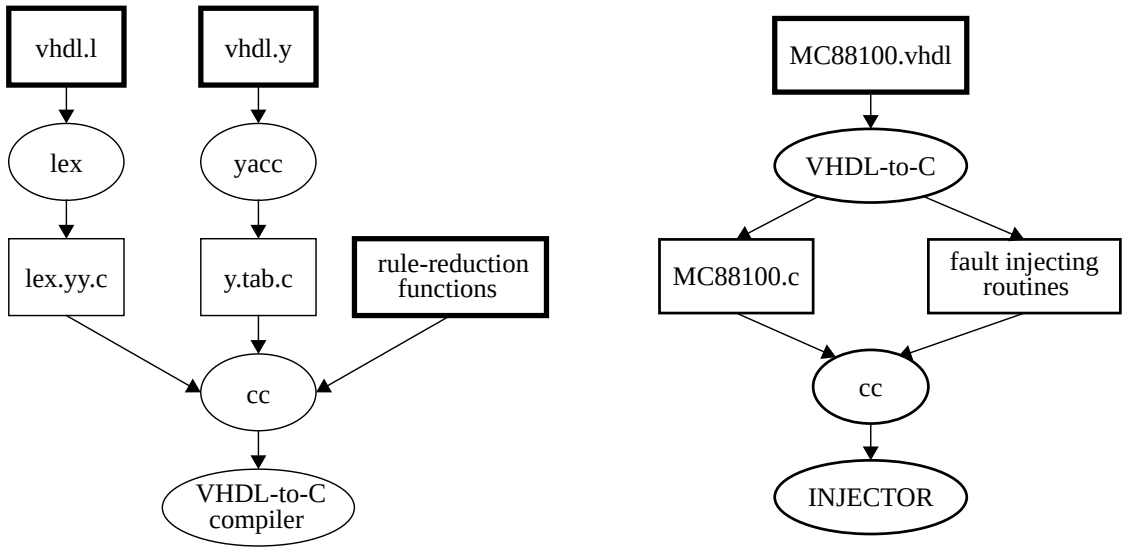


Fig. 3: The generation and the usage of the VHDL-to-C compiler

Items in the chart denote files; rectangles mean text files, while ovals mean executable ones. Text files with a thick border indicate manually provided scripts

- *Data storage function*: Custom error patterns are modulo 2 added to the register, similarly to the error level modelling, except that injections between microinstructions are performed, too.
- *Data transfer function:* The method is the same as for the data storage function, but limited to the destination registers of recent data transfers.
- *Data manipulation function:* since there is no general fault model defined for this function, manipulation functions are not replaced with faulty ones. Error effects can be approximated by data transfers faults with a user defined probability distribution.
- *Register decoding function:* A distinguished function “Decode” is substituted with another function that returns random wrong values, according to the general decoder fault model.
- *Instruction decoding and execution:* Functional microorder procedures are skipped or additionally executed.

User interface

The injector is implemented as a C function under UNIX in the form of
int Injector(pid, customisation_mask, ptr_results_info, last_sig)

This function can be linked to the software fault injector after customizing its behaviour by the second argument. This allows the selection of a fault class to be injected as well as additional confinements within this class (e.g. for the register decoding function, injection can be limited to the “No register is selected, ZERO constant is read” fault subtype). The number of faults and the fault selection strategy (deterministic/random) can be set, too.

Its operation resembles to a **ptrace(SINGLE_STEP, pid, 1, 0)** call. It simulates the execution of the next instruction, while injecting custom faults into it. The injector can return five major result types:
- *successful fault injection*, there is an impact of the injected fault on some register;
- a fault has been injected, but there is *no impact on the register content*, due to fault masking in the later phases of the simulated instruction;
- would-be *exception* during simulation;
- the instruction is *unimplemented*, therefore not simulated;
- some error or unexpected event has occurred, e.g. the observed program has been killed before the end of the simulation.

Information about exceptions or the differences in the register content (the syndrome) is available in the memory area pointed to by **ptr_results_info**.

Portability issues

When speaking about portability, the following questions must be considered: what processors can be described with this VHDL subset and description style? Is it sure that they can be compiled the same way as in the case of the MC88100? What particularities of the environment have been exploited?

General purpose microprocessors can be described as far as they conform to the discussed functional model. It involves that the majority of processor types can be modelled with the exception of only a few instructions. RISC processors are easier to describe from the sequential point of view than more complex CISC processors. Regarding the compilation and environmental principles, those processors can be simulated using this methodology that support operating systems with ptrace-like register access and single step execution.

Some disadvantages and limitations originate from the current implementation. First, compiling to a sequential and procedural language has side effects: parallel effects of faults cannot be satisfactorily modelled. Two virtually parallel microorder procedures are executed sequentially during simulation, therefore if they have an identical destination register, the effect of the microorder executed later will dominate. A simulator offering a support of the resolution of signal assignment conflicts would solve this problem. Another drawback of the method is its dependency on the refinement of the model from the processor user manual to the formal description. However, a wrong assignment of operations to structural units has only a slight impact on the simulation results, when transient faults are modelled only.

The injector is unable to internally determine the probabilities of the individual fault classes; a non-uniform distribution must be provided by the user.

There are several directions the methodology and the compiler could be improved toward. An open type set would be more comfortable for the user. Permanent faults could be the subject of simulation, although it would require much more detailed information about the structure of the architecture, and we should assure that the same fault injecting mechanisms are activated.

Measurement environment

Our primary goal was the comparison of the conventional error-level injecting techniques with the more detailed VHDL-based models.

The fault injecting experiments were organized as follows: The benchmark application launched many instances of the observed process.[2] Breakpoints were inserted randomly, one into every instance. With the use of system ptrace services, the content of the registers was easily observable and controllable. The next instruction after the breakpoint was executed twice. In the first run, the fault-free, reference register image was computed. Prior to the second run, the initial register content was reconstructed, and a faulty instruction was simulated. A record of the syndrome (the difference between the two register images) was stored for subsequent statistics (short-term results of the experiment). Then the observed program was continued without any further breakpoint in order to determine the long-term effects of the injected fault, provided that it did not involve a fatal error, like an exception already during the faulty instruction. Later, error messages were added to a log file.

Short-term results

The syndromes given back by the injector function were examined to reveal the probability distribution of resulting register error types.

Data storage faults were obtained by the injection of single-bit faults into a random register. The histogram in Fig. 4 shows the frequency of errors against their Hamming weight in one register.

In the traditional assumption on single bit errors, the corresponding histogram would contain only a single spike at the Hamming-weight of 1. The vast majority of injected one-bit errors indeed cause a one-bit difference in a register in the end of an instruction. It proves the validity of traditional fault injectors for faults of the data storage class. However, a low percentage of experiments ended in exceptions or discrepancies in multiple registers, even in this simplest case. This fraction originates from

[2] usually self-test and benchmark programs

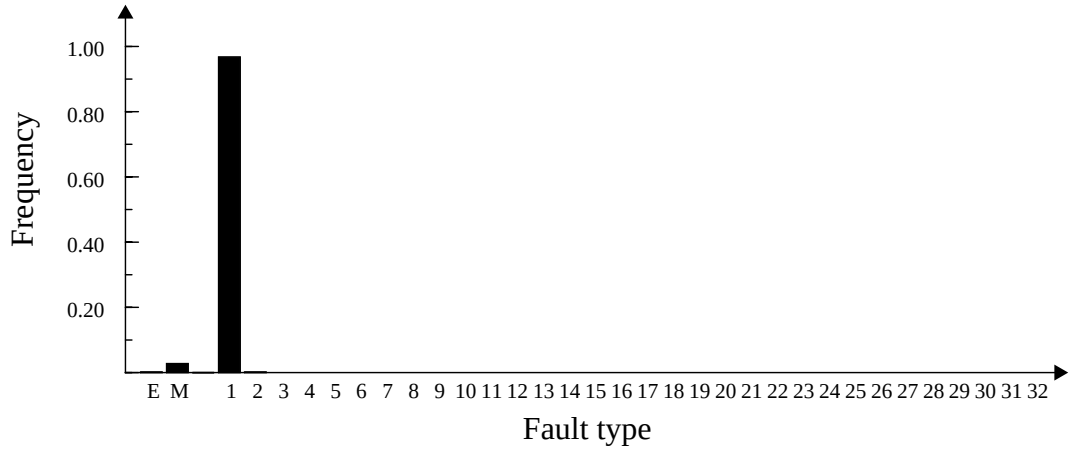


Fig. 4: Effect of storage faults

Legend: E=would-be exception during simulation; M=multiple-register error;
1-32=error in one register with a given Hamming weight

faults injected into a source register of a subsequent microoperation.

Fig. 5 depicts the results of *data transfer fault* injection. Although one-bit error results still have an

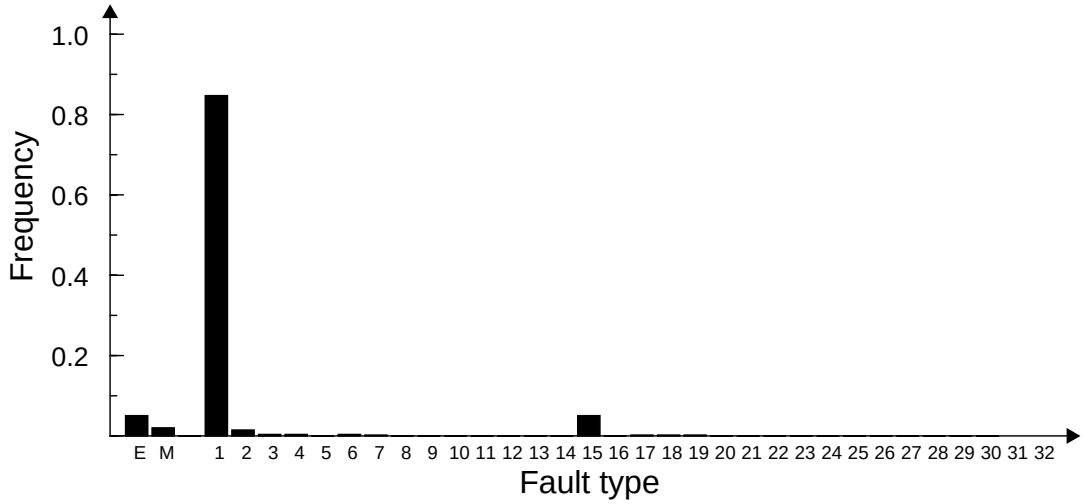


Fig. 5: Effect of data transfer faults

essential majority, we should recognise that other errors already have a perceptible frequency that must be taken into account when designing systems for transfer fault tolerance. This result is not a surprise: transfer faults are more serious than data storage faults, because the content of the transfer destination register is likely to be used for further processing, e.g. for data manipulation or address calculation.

Neither conventional random single-bit, multiple-bit, nor word injection models the effects of *register decoding faults* properly (Fig. 6). If one-bit injection was a good model, a histogram like Fig. 4 would be obtained. For multiple-bit or random word errors, we should have obtained a histogram following a p parameter binomial probability distribution (p = 1/2 for random word errors). The same consequences apply for *microorder faults* (Fig. 7), practically uncovered by conventional error-level fault models.

Long-term results

Fig. 8 and Fig. 9 show the different integrated error latency distributions of single-bit storage faults and additional microorder faults. The x-axis shows the number of instructions executed after the fault injection. The y-axis shows the percentage of special failure types occured in the interval from 0 until x instructions.

The different failure types are:

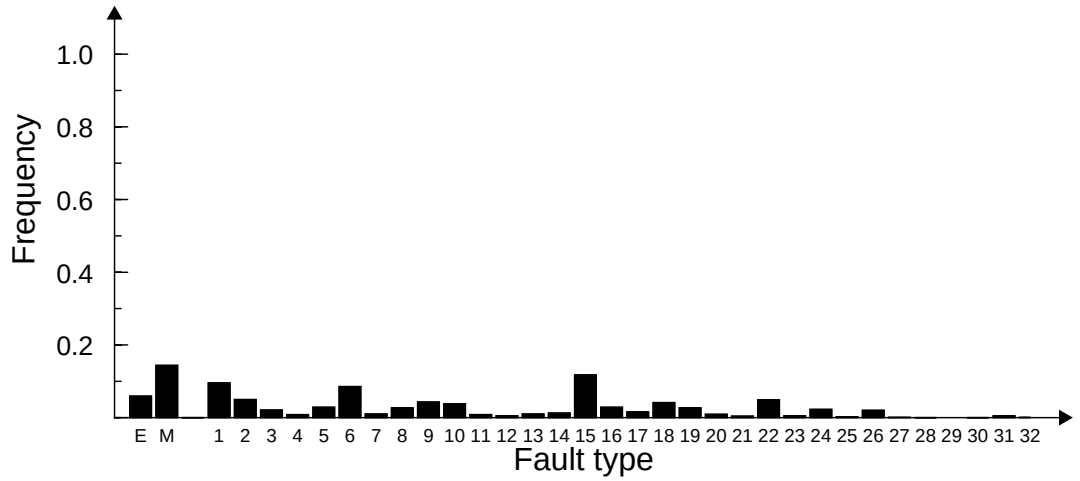


Fig. 6: Effect of register decoding faults

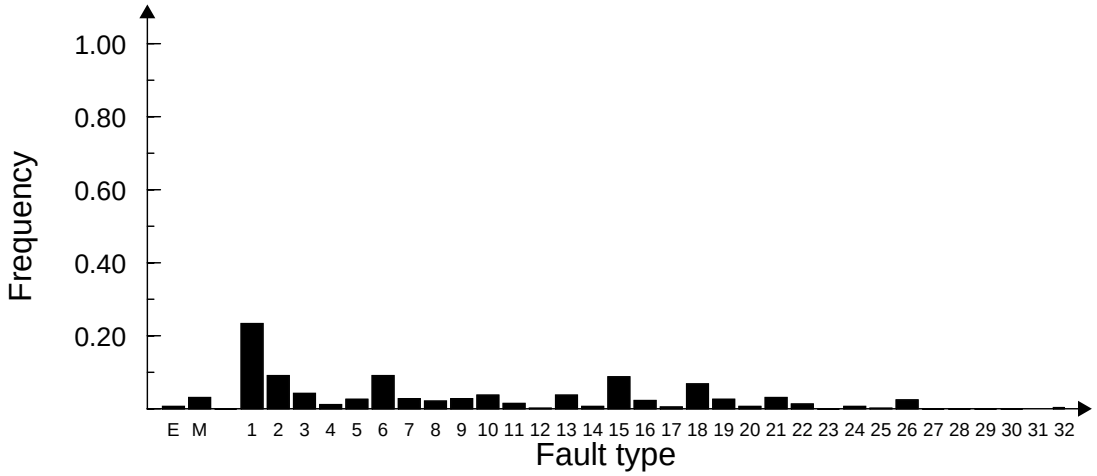


Fig. 7: Effect of microorder faults

- Segmentation violation: The processor tried to access memory cells outside his address space;
- Bus error: The processor tried to access a word in memory with illegal word alignment;
- Division by zero: The processor tried to divide a number by a zero value;

The rest of the experiments gave correct or incorrect program results. The processes terminated with the normal exit() statement. These experiments are omitted in Fig. 8 and Fig. 9, because there exists no error latency in these cases.

The latency of single-bit storage faults (Fig. 8) is a distribution of the form $K_1 e^{-K_2 x}$. K_2 is ≈ 0.036 for all different failure types. K_1 depends on the type of the failure. Because it is nearly an exponential function, one can say that the probability for a process to die due to an erroneous bit in one of the processor registers is nearly the same at every instruction.

This is not the case in the second experiment as shown in Fig. 9. The probability of a process to die is much higher at the next few steps than later. One explanation may be the following: If instructions are executed in a faulty way by injecting additional microorder faults, mostly data used by the next few instructions is destroyed. In the first experiment (single-bit storage faults), only a part of the processor registers contains data used later.

Conclusion

We can conclude finally that conventional injecting techniques provide realistic fault sets only for the data storage fault class, and they roughly approximate the effects of data transfer faults. However, error-level faults are unacceptable models of register and instruction decoding faults, for which alterna-

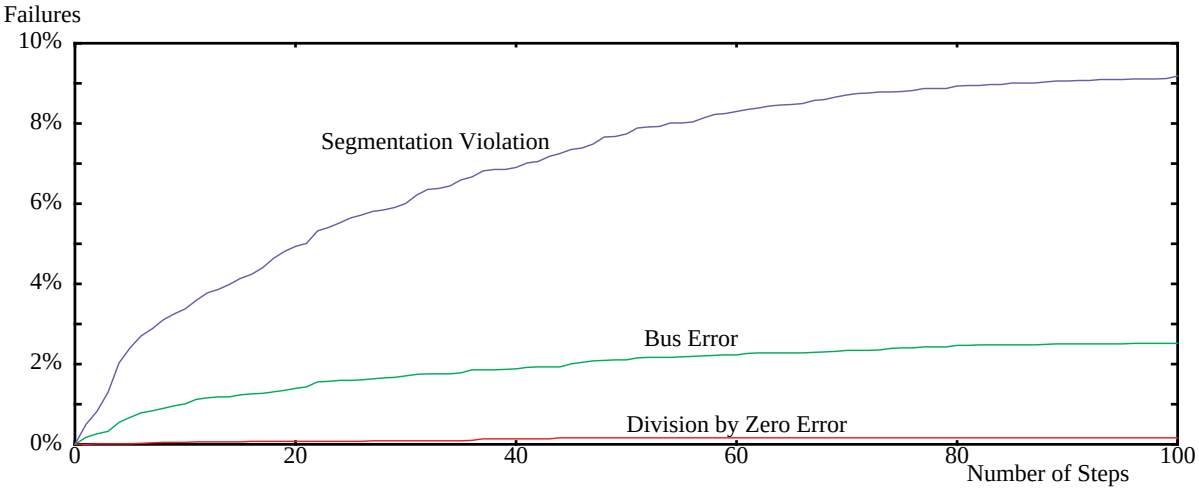


Fig. 8: Latency of single bit storage faults

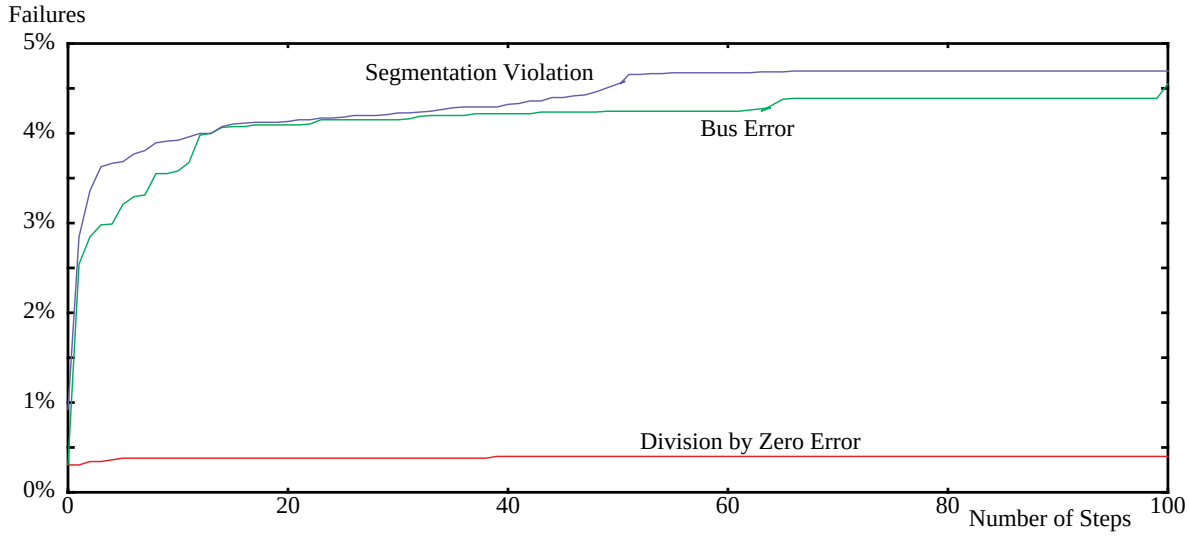


Fig. 9: Latency of additional microorder faults

tive methods, like the simulator presented here, must be used.

References

[1] S. M. Thatte, J. Abraham: Test Generation for Microprocessors. IEEE Transactions on Computers, Vol. C-29, No.6, June 1980

[2] D. Brahme, J. Abraham: Functional Testing of Microprocessors. IEEE Transactions on Computers, Vol. C-33, No.6, June 1984

[3] R. Lipsett, C. Schaefer, C. Ussery: VHDL: Hardware Description and Design, Kluwer Academic Publishers, 1989.

[4] IEEE Standard VHDL Language Reference Manual. IEEE, Inc., New York, NY, March 1988

[5] MC88100 RISC microprocessor. User's manual. Prentice Hall, 1990.

[6] Dal Cin, M. et al.: "Fault Tolerance in Distributed Shared Memory Multiprocessors", *in: A. Bode, M. Dal Cin (eds.): Parallel Computer Architectures - Theory, Hardware, Software, Applications*, Springer Lecture Notes in Computer Sciences 732, 1993, pp. 31-48