Systematically Generated Diversity to Improve Online Hardware FaultDetection and Validation of Results

Heidrun Dücker

Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe
Postfach 6980, W 7500 Karlsruhe 1, Germany
Tel. ++49-721-608-4353, Fax: ++49-721-370455, mail: duecker@ira.uka.de

Abstract

Existing design diversity, which is generally used to detect software faults, enables hardware fault detection without any additional measures, but the obtained hardware fault coverage is insufficient. To improve the hardware fault coverage we present a method that systematically transforms every instruction of a given program into a modified instruction (sequence), keeping the algorithm fixed. This transformation bases on a diverse data representation and accompanying modified instruction sequences. The original and the systematically modified variants of a program are executed sequentially. Afterwards both results are compared on-line to detect hardware faults. First examinations of the hardware fault coverage have shown that the fault coverage of design diversity can be improved by additionally using systematically generated diversity.

Keywords: Diversity, hardware faults, software implemented fault tolerance, testing, dependability.

1 Introduction

Safety-critical applications need dependable hardware and software. Dependable program execution essentially requires hardware fault detection or hardware fault tolerance. Instead of the common approach of structural duplication, we present an approach based on systematically generated diversity: one program is run multiple on the same system and the computed results are compared. This method is not suitable for complete hardware test, but it allows validation of results.

While running the same program twice transient faults can be detected easily. Permanent faults require additionally a transformation of the program to obtain different results with a high probability. Such a transformation could be gained by design diversity or by systematically generated diversity.

If design diversity is used, different design teams evolve different solutions of the same problem to get diverse implementations. If systematically generated diversity is used, one design team evolves one solution of the problem and the second variant is generated by a precompiler by using systematic transformations. Instead of an original instruction an accompanying instruction sequence, applied to operands given in the diverse data representation, will be executed. This new instruction sequence calculates the original result in the diverse data representation. If every instruction of a program is modified with respect to the diverse data representation, a variant is obtained that calculates the results with the same algorithm but with different instruction sequences and different data flow.

Data complementation has been suggested as a diverse representation promising good fault detection and small additional effort. A prototype of a precompiler has been implemented, which modifies a given program with respect to a complementary data representation.

Section 2 generally describes how hardware fault detection works by using software diversity. In section 3, the method for systematically generation of diversity and realization of the method is described. In section 4 some experimental results are presented.

2 Theoretical principles

Hardware and software faults may be detected by the following tests:

Absolute tests: Concerning some consistency conditions a computer system or data could be examined on-line by absolute tests. Checking the address range or kernel calls also belongs to the class of absolute tests as application-dependent tests of the results do. The fault coverage of absolute tests depends on known conditions. Only faults that break the conditions could be detected.

Relative tests: If a function is executed for several times the computed results may be compared by a relative test. The results could be calculated either parallel or serial. With a relative test any type of independent faults could be detected, but the number of detected or tolerated faults depends on the number of redundant calculations of the functions.

A widespread method to detect software design faults is **diversity**, where different variants of a program are used which comply with the same specification. Later the distinct variants are executed either in parallel or serial. Parallel execution requires multiple hardware, but scarcely time redundancy. Much time redundancy but no structural redundancy is required by serial execution.

We distinguish between design diversity and systematically generated diversity:

If **design diversity** is used, different design teams develop different variants of a program to minimize the probability of common design fault in the different variants. Sometimes the teams make an arrangement with regard to specification method, programming language, compiler or type of the used processor. The most important arrangement is the one about different algorithms which will be implemented.

In opposite to design diversity, **systematically generated diversity** produces the different variants from a given program without changing the implemented algorithm but through computable modifications of the program or data flow. There are different possibilities to generate systematically diverse variants:

- The exchange of the registers is an easy way for systematically generated diversity. Then different variables of the algorithm may be affected by the stuck-at-faults of the registers, and these faults could be detected.
- An other way is to look for sequences of instructions that may be parallelized. Then the execution order of these sequences could be changed. The different execution order causes a different order of interim results and therefore especially allows a detection of data dependent faults.
- New diverse sequences of instructions can be developed with some knowledge about the semantic of often used sequences of instructions. Considering the application dependent features, a precompiler could replace the original sequence by the new sequence.
- Using a different data representation enables systematically generation of diversity. The diverse data representation could be realized with a bijective mapping on the data. Additional modifications of the used instructions are forced by the diverse representation of data. That means: instead of a given instruction an accompanying sequence of instructions is executed, that calculates the original result in the diverse representation. If all instructions of a given program are transformed concerning a given mapping, a variant is generated that calculates the results with a different data flow but without changing the implemented algorithm.

The first three variants of systematically generated diversity make use of modifications of the program flow without taking data and its representation into account. The last variants transforms the data of the definition range of the instructions to the range of modified sequences of instructions provided that transformed data, modified sequences of instructions, and retransformed data calculate the same result as the original variant. A precompiler for systematically generation of diverse variants of a given program can be realized for all explained possibilities, because they facilitate computable transformations.

2.1 The principle of hardware fault detection using diverse software

2.1.1 Process of fault detection

If a system is given that runs the software once (simplex system), hardware faults that affect computed results may be detected only with a concurrent absolute test. Faults that cause an exchange of the results within the given range of an absolute test, cannot be detected in such a system.

On virtual-multiple-systems diverse variants are run serially on one hardware. As diverse variants do not use registers and instructions in the same way, hardware faults could affect different parts of the calculation. Many faults could be detected by the absolute tests during the additionally executed variants or by the following relative tests.

With two variants on one hardware, a fault detecting system is obtained, that detects hardware and/or software faults depending on the diversity employed:

- If systematically generated diversity is used, only hardware faults could be detected because design faults are duplicated. The afford in time that is required to detect hardware faults is very high, since the program execution is duplicated. Therefore other known methods may be more advantageous.
- If design diversity is used ,hardware and software faults could be detected, but the hardware fault coverage is low [EHNi 90].
- A combination of systematically generated and design diversity preserves software fault detection and increases hardware fault detection. Figure 1 shows a virtual duplex system with two variants, realising design diversity. One of the two variants additional is modified systematically to increase hardware fault coverage.

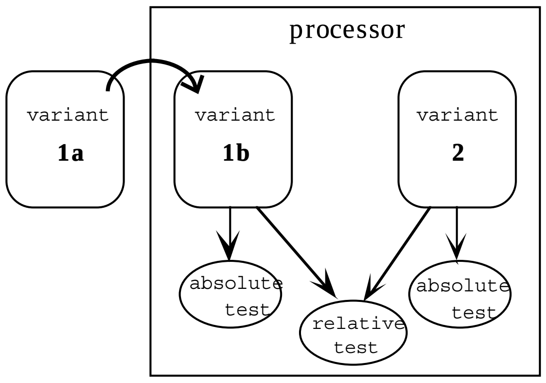

Figure 1: Diagram of a virtual duplex system of systematically generated (variant 1a and 1b) and design diversity (variant 1 and 2)

2.1.2 Fault model

The fault model used is a single-stuck-at fault model for the processor hardware. Therein only processor dependent faults have been considered, because memory faults could be detected by coding methods. Accompanying to the family of Motorola 68000 the processor dependent faults can be subdivided into three types of faults:

Bus faults:	a single line is stuck at 0 or 1.
Register faults:	a single bit is stuck at 0 or 1.
Instruction faults:	a single instruction of the instruction set is wrongly executed, i.e. instruction J is executed instead of instruction I. = a fault in the control unit causes a wrong execution of an instruction.

The fault coverage is recorded by a software-implemented fault injector [Hinz 89] which is realized for the processor M68000. This fault injector simulates the modelled fault effects for every instruction by executing the program in a single step modus. After the complete execution of both diverse program variants it checks whether the fault could be detected or not.

The following classes for the results of the examination exist:
1) correct
2) faulty; detected by an absolute test
3) faulty; detected by the relative test
4) dangerously faulty; not detected, as the number of identically faulty results is greater than the number of correct results

In a system that runs the application once, Class 3 does not exist. Class 4 always takes place if a wrong result is calculated.

Besides the detection of software faults the use of design diversity facilitates a low detection of hardware faults, which results in a small reduction of the appearance of the situation ´dangerously faulty´. The additional use of systematically generated diversity makes it possible to detect hardware faults that initially leaded to the situation ´dangerous faulty´.

3 A technique for systematically generation of diversity

3.1 The basic principle

The method presented here is based on modification of the data representation. Since a divers data representation also affords a modification of all instructions, that may be executed, new sequences of instructions have to be generated, that calculate the result of the original instruction in the modified representation. This generation has to be made regarding the used representation of data as well as to the used processor. Concerning a definite representation R of data there exists the following relation between the instructions of the diverse variants:

Definition 1:
Given a processor P with a range D of representable data and an instruction set F $= \{f_i: DxD \rightarrow W \subseteq D\}$. For a representation R of data there exists a bijective, computable mapping $r: D \rightarrow D$ so that for every instruction f_i there exists a sequence s_i of instructions of instructions from F with the following characteristics:

$$s_i : r(D) \text{ x } r(D) \rightarrow r(W)$$

$$\text{with} \quad s_i(r(a) , r(b)) = r(f_i(a,b)) \quad \text{and} \quad a,b \in D.$$

Accompanying to the instruction f_i, the diverse sequence of instructions regarding to the representation R is defined by:

$$r^{-1} \circ s_i \circ r$$

A modification of both variants is an other method to get comparable sequences of instructions: $s_i \circ r$ and $r \circ f_i$.

To an instruction f that calculates the result f(a,b) from the operands a and b the accompanying sequence of instructions needs the following three steps to calculate the result r(f(a,b)) (see fig. 2):

1. Transformation of the operands with the mapping r

2. Execution of a special sequence of instructions s that computes the transformed operands into the transformation of the result.

3. Retransformation of the result with the inverse function r^{-1}.

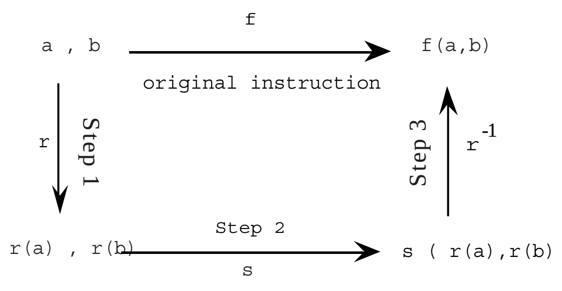

Figure 2: Transformation of an instruction

In a program every instruction is modified with regard to the given mapping r. Thereby Step 3 of the instruction f_i and Step 1 of the instruction f_{i+1} could be dropped (see Fig. 3) because these steps only transform the intermediate variables from one representation into the other. This is valid because the retransformation of r(x) is given by $r^{-1}(r(x))$ and for the additional transformation of this value holds:

$$r(r^{-1}(r(x))) = r(x)$$

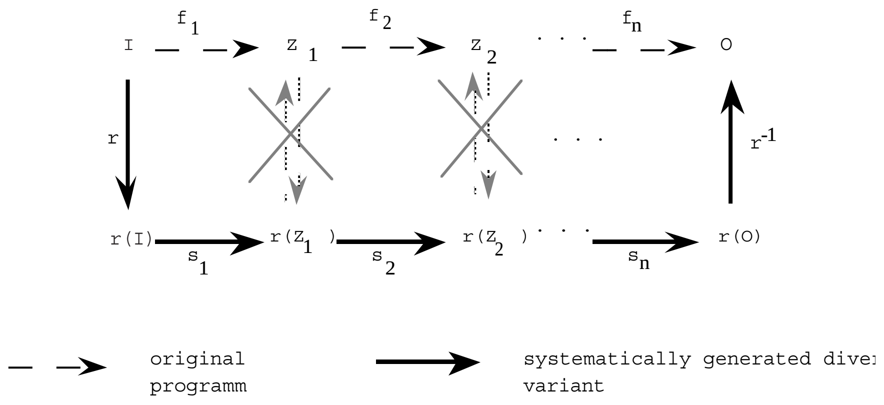

Figure 3: Generation of a systematically generated diverse variant

Fig. 3 shows the transformation of a program into an accompanying systematically generated diverse variant. An original program calculates the output O from the input I with the intermediate variables Z_i of the instructions f_i. Then the systematically generated diverse variant calculates the output with the intermediate variables $r(Z_i)$ while using the modified sequences of instructions s_i. The transformations of data between the sequences s_i (Step 3 and Step 1) could be dropped.

3.2 Implementation

3.2.1 Choice of the mapping r

In principle every bijective mapping r: D $\rightarrow$ D that allows a modification of every instruction of an instruction set of a given processor could be used to implement systematically generated diversity. But the efficiency of the method requires that the mapping r must be easy to implement and that the sequences s_i could be realized efficiently. In addition the use of the systematically generated diversity accompanying to this mapping r must improve the detection of hardware faults.

The good detection of real faults of the stuck-at fault model should be picked up in the choice of the used mapping. A method that is used in memories complements the stored data to distinguish permanent and transient faults. An other advantage of data complementation is that the range of data is not modified as it happens with the one-bit-shifting of the data that Hahn and Gössel [HaGö 91] use. Furthermore for the mappings r and r^{-1} data complementation could be realized easily.

Two methods are known to complement binary values: complement-on-one and complement-on-two. Systematically generated diversity with data complementation requires an efficient realization of the sequence s for every instruction f of the given instruction set. To be able to find a good decision the instruction classes that manipulate data must be examined. The most important classes are the arithmetic and logical instructions.

According to the complement, logical instructions could be transformed with the rules of de Morgan. These rules transform disjunction in the following way:

$$a \vee b = \overline{\overline{a \vee b}} = \overline{\overline{a} \wedge \overline{b}}$$

This formula could be applied to the logical instruction *OR a,b*. To this instruction the diverse instruction is given through the instruction *AND* that calculates the inverted result of ¬c = OR a,b from the inverted operands ¬a and ¬b:

$$\neg c = \mathrm{OR}\ \neg a, \neg b$$

The rules of de Morgan could be applied to all logical instructions of a processor in analogy (see Table 1):

transformation	$f_i \rightarrow r^{-1}(s_i(r(a,b)))$

$$\begin{array}{lllll|lll}
a \vee b & = & \overline{\overline{a \vee b}} & = & \overline{\overline{a} \wedge \overline{b}} & \text{or } a,b & \rightarrow & \neg(\text{and } \neg a,\neg b) \\
a \wedge b & = & \overline{\overline{a \wedge b}} & = & \overline{\overline{a} \vee \overline{b}} & \text{and } a,b & \rightarrow & \neg(\text{or } \neg a,\neg b) \\
a \not\equiv b & = & \overline{\overline{a \not\equiv b}} & = & \overline{a} \not\equiv \overline{b} & \text{xor } a,b & \rightarrow & \neg(\neg(\text{xor } \neg a,\neg b))
\end{array}$$

Table 1: Correlation between the logical instructions and the complement-on-one

Regarding the arithmetic instructions the inversion of the all bits of the value x (= complement-on-one) causes the following modification of the value:

$$\overline{x} = -1 - x.$$

Concerning the arithmetic instruction "signed division" the diverse sequence of instruction must be generated according to

$$\overline{\left(\frac{x}{y}\right)} = (-1) - \frac{x}{y} \overset{!}{=} \frac{(-1-x)}{(-1-y)}_{\text{diverse}} = \frac{\overline{x}}{\overline{y}}_{\text{diverse}}$$

The modified sequence of instructions $s = \frac{(-1-x)}{(-1-y)}_{\text{diverse}}$ could be calculated in the following way:

$$\overline{\left(\frac{x}{y}\right)} = -1 - \frac{x}{y} = -1 - \frac{-1-\overline{x}}{-1-\overline{y}} = -1 + \frac{1}{-1-\overline{y}} + \frac{\overline{x}}{-1-\overline{y}} =$$

$$= -1 + \frac{1}{-1-\overline{y}} + \frac{1}{\frac{-1}{\overline{x}} - \frac{\overline{y}}{\overline{x}}} = -1 + \frac{1}{-1-(-1-y)} + \frac{1}{\frac{-1}{-1-x} - \frac{-1-y}{-1-x}}$$

Therefore $\frac{\overline{x}}{\overline{y}}_{\text{diverse}}$ has to be replaced by $-1 + \frac{1}{-1-(-1-y)} + \frac{1}{\frac{-1}{-1-x} - \frac{-1-y}{-1-x}}$

to fulfil the required equation. This calculation is very costly. In addition this calculation is not defined for x = –1 as well as for y = 0, while the original division is not defined for y = 0 only .

This shows that the complement-on-one is not an efficient realization for data complementation for all instructions. For arithmetic instructions the complement-on-two is a more efficient realization, because there exists a similar relation between the instructions and the complementary representation of data as shown for the logical instructions and the complement-on-one (see Table 2):

transformation			f_i	$\rightarrow$	r^{-1} (s_i (r (a,b)))
a + b	=	-((-a) + (-b))	add a,b	$\rightarrow$	–(add –a,–b)
a – b	=	-((-a) – (-b))	sub a,b	$\rightarrow$	–(sub –a,–b)
a · b	=	(-a) . (-b)	muls a,b	$\rightarrow$	–(–(muls –a,–b))
a / b	=	(-a) / (-b)	divs a,b	$\rightarrow$	–(–(divs –a,–b))

Table 2: Correlation between the arithmetic instructions and the complement-on-two

At the other hand the complement-on-two is not suitable for logical instructions. Therefore the mapping r realizes the complement-on-one as well as the complement-on-two.

3.2.2 Optimization of the implementation

3.2.2.1 Updating the status register

After the selection of the diverse representation to all instructions of the instruction set of the used processor, the modified sequences must be generated. By the examination of the instructions it has to be recognized that a single instruction not only affects the explicitly given operands but also the user byte of the status register (see Fig. 4).

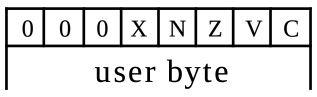

Figure 4: Structure of the user byte of the status register of the processor M68000

The bits 0 - 4 of the user byte, also called condition-code-register, are reserved for condition flags, bit 5 - 7 always contain the value 0. The flags are automatically set or cleared in dependency on the operands or on the result of an instruction. They are used for conditional branches. It depends on the executed instruction which flags are modified. Not every instruction changes every flag.

In a systematically generated diverse variant of a program the instructions of the original program are replaced by new sequences of instructions. The bits of the status register should be set in the complement-on-one. Some of the diverse sequences automatically fulfil this requirement. Some sequences change the values of the flags in such a way, that there exists no relationship between the status register of the original program and of the modified variant. A following instruction, e.g. a branch, which calls up one or more flags needs a relationship between the original and diverse variants to execute this instruction correctly. That is the reason for updating the flags of the status register.

Therefore diverse sequence of instructions must take into account the explicitly given operand and the implicitly used status register. Updating of flags requires much time on heat (runtime), but in principle it is needed only if afterwards the status register is called up.

For reduction of the runtime-overhead a criterion that indicates when updating is necessary has to be generated. If every executed instruction would change every flag, a single look on the following instruction could tell if the flags have to be updated or not. Unfortunately the most instructions only change some of the flags. Therefore for every flag it has to be stored which instruction changed it the last time. If all flags that are changed by an executed instruction are modified by the following instruction before an instruction is executed that calls up one of the flags, updating of the status register is not necessary for this instruction.

The most used instructions that call up the flags are conditional branches. During the generation of the diverse variant it is not known which way in the program flow the execution will take. Therefore the status register has to be updated before every conditional branch for both possibilities: fulfilled and not fulfilled condition. If the precompiler is implemented as a one-pass-compiler it is not known which instructions will be executed after a branch, because the transformation is done serially through the program listing. In this case the status register has to be updated before every branch.

3.2.2.2 Switching between complement-on-one and complement-on-two

An efficient realization of a precompiler for systematically generating diversity requires the application of the complement-on-one and the complement-on-two (see Section 3.1). Dependent on the used class of instructions the used type of diverse representation of data is changed between the complement-on-one and the complement-on-two. Switching between the two types of complement could be implemented easily by the addition or subtraction of "1". Nevertheless, often switching between two different representations of data leads to an enormous waste of time.

Most instructions cannot be modified by one type of complement, but one type normally is more efficient than the other. Regarding the efficiency of a transformation it has to be taken into account that switching between the two types of complement also needs time. Therefore for some instructions the modification in the non-efficient type of complement is more efficient than the switching between the types of complement together with the efficient transformation:

Example:

The transformation in the complement-on-one of a single addition of integers that are enclosed into logical instructions is more efficient than changing and rechanging of the representation of data and the efficient transformation (see Table 3). The transformation in the complement-on-one only needs one additional addition for compensation. On the other hand changing into the complement-on-two and rechanging into the complement-on-one needs two additions as well as a subtraction.

f	transformation in the complement-on-one	switching	transformation in the complement-on-two
add a,b	add ¬a,¬b	add #1,¬a	
	add #1,¬b	add #1,¬b	
			add –a,–b
		sub #1,–b	

Table 3: Comparison of the different transformations of instruction "addition"

The type of complement which has to be used for the most efficient realization for the given instruction, could be derived from the context in which the instruction is embedded.

- If a single instruction is embedded in a sequence of instructions that are transformed more efficiently with the non-efficient type of transformation for this special instruction (see Fig. 5.b), then the transformation with the non-efficient type of complement could be a more efficient solution for the whole program (compare Table 3).

- Sequences of instructions that are efficiently transformed with the same type of complement, are transformed usefully with the efficient type of complement. If a new sequence follows that is transformed efficiently with the other type of complement, then the switching is an efficient solution. The instructions for switching only take a small share of all instructions, because they are executed only if a change of sequence appears that use different types of complement (see Fig. 5a).

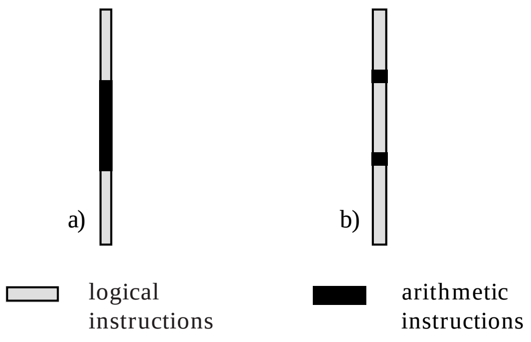

Figure 5: Diagram of sequences of instructions:

a) big sequences of instructions with the same representation of data

b) sequences of instructions with the same representation of data, that are interrupted by single instructions that are normally transformed in the other representation

Dependent on the context of an instruction it can be decided if use of the current representation of data should be used or if switching is the more efficient solution. Variants can be generated to the same given program while using the complement-on-one and the complement-on-two several diverse. The goal of switching is to generate the most efficient variant with regard to runtime.

Definition 2:
All possible diverse variants based on a given program P with the instructions f_i, $i \in \{1,...,n\}$, could be obtained from a transformation-graph TG_P.

Node $N_{0,i}$, $i \in \{1,...,n\}$, of the transformation graph corresponds to the internal values of the original program before executing the instruction f_i.

Node $N_{d,i}$, $d \in \{1,2\}$ and $i \in \{1,...,n\}$, of the transformation graph corresponds to the internal values of a diverse variant in which the last instruction f_{i-1} is transformed with the complement-on-one (d=1) or with the complement-on-two (d=2), respectively.

Edges $s_{d,i}$, $d \in \{1,2\}$ and $i \in \{1,...,n\}$, of the transformation graph corresponds to the modified sequences of instructions in the complement-on-one (d=1) or in the complement-on-two (d=2) accompanying to the instruction f_i, respectively.

Edge $t_{d,i}$, $d \in \{1,2\}$ and $i \in \{1,...,n\}$, of the transformation graph corresponds to the transformation of the operands of the instruction f_i from the complement-on-one into the complement-on-two (d=1) and vice versa (d=2).

Figure 6 shows a transformation graph as defined in definition 2.

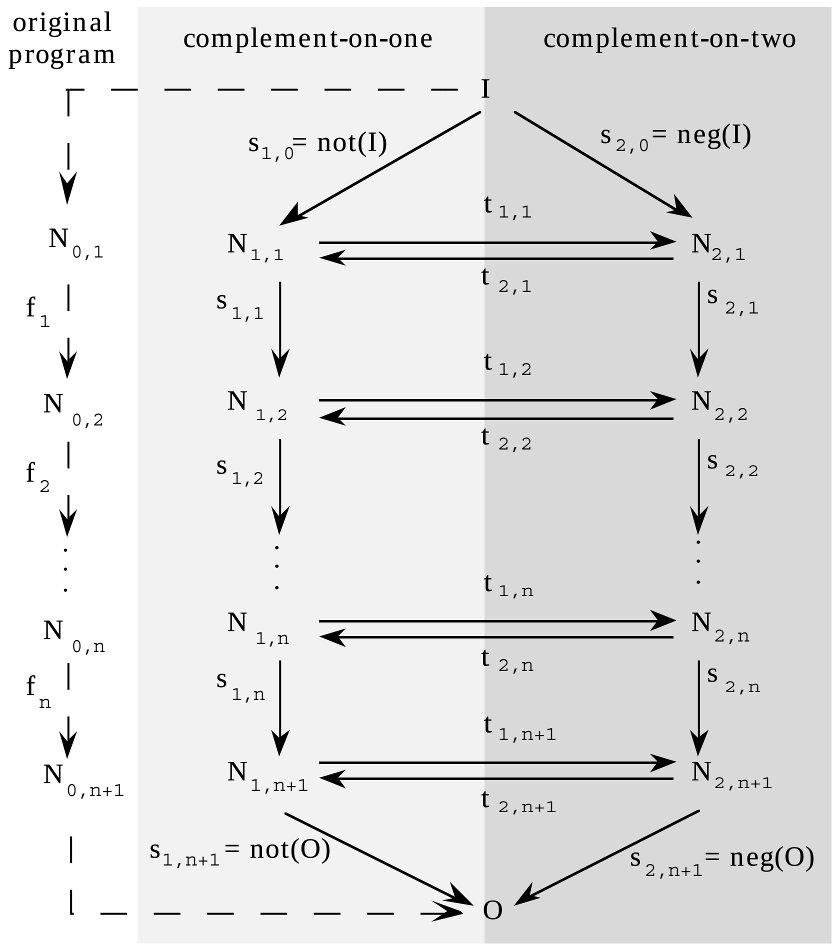

Figure 6: Transformation graph TG_P of a program P with input I and output O

An assembler program of a diverse variant corresponds to a simple path through the transformation graph from I to O in which every node of the graph is joined maximally once. Such a path through the transformation graph is a subgraph of TG_P and is called TG_P' in the following. This subgraph TG_P' includes exactly one modified sequence of instructions $s_{d,i}$ and at most one transformation sequence $t_{d,i}$ for the operands at every level i (see Fig. 7).

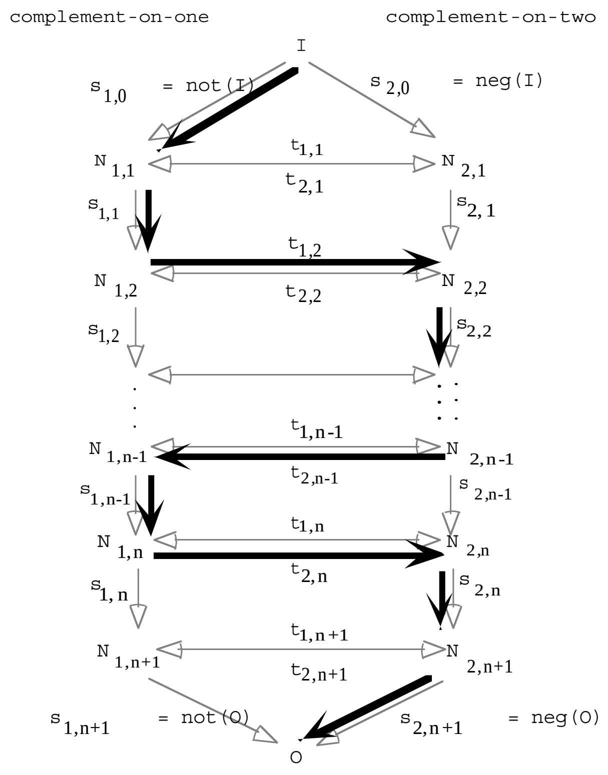

Figure 7: Diagram of a subgraph TG_P'

For an original program P with the instructions f_i, $i \in \{1,...,n\}$, a diverse variant is required, that is systematically generated by a precompiler which bases on the complementary representation of data and takes minimal runtime. With the following transformation function a formula can be generated that gives the effort of a diverse variant.

Definition 3:

The transformation function trans(i) indicates the representation of data (complement-on-one or complement-on-two), corresponding to which the instructions f_i are transformed. This function is defined in the following way:

$$\text{trans}(i) := \begin{cases} 1 & \text{if } s_{1,i} \text{ is an edge of } TG_P' \\ 0 & \text{if } s_{2,i} \text{ is an edge of } TG_P' \end{cases} \qquad i \in \{0,...,n+1\}$$

$$\text{trans}(-1) := \text{trans}(0)$$

Theorem:

Let E(x) be the number of clock cycles that are used for the execution of a sequence of instructions x and let TG_P be the transformation graph to a program P with n instructions. The effort $E(TG_P')$ of a diverse variant accompanying to a subgraph TG_P' is calculated by:

$$E(TG_P') = \sum_{i=0}^{n+1} \{(1-trans(i))\cdot[trans(i-1)\cdot E(t_{1,i}) + E(s_{2,i})] + trans(i)\cdot[[1-trans(i-1)]\cdot E(t_{2,i}) + E(s_{1,i})]\}$$

Proof:

The proof is divided into two parts:

1. Without regarding the retransformation of the results, the effort of a diverse variant of a program P with n instruction is calculated by:

$$E'(TG_P') = \sum_{i=0}^{n} \{(1-trans(i))\cdot[trans(i-1)\cdot E(t_{1,i}) + E(s_{2,i})] + trans(i)\cdot[[1-trans(i-1)]\cdot E(t_{2,i}) + E(s_{1,i})]\} \qquad [3.1]$$

2. The effort of the retransformation is calculated by:

$$(1-trans(i))\cdot[trans(i-1)\cdot E(t_{1,i}) + E(s_{2,i})] \quad + \quad trans(i)\cdot[[1-trans(i-1)]\cdot E(t_{2,i}) + E(s_{1,i})]$$

Therefore follows: $E(TG_P') = E'(TG_P') + E(retransformation)$

The first part is proven through induction over the number of instructions n of the original program P:

With regard to the subgraph TG_P' and the accompanying diverse variant $E'(TG_P')^i$ denotes the effort of the modified sequences of instructions accompanying to the instructions $f_1, ..., f_{i-1}$ of the program P.

Basis Step:

To a minimal program (n=1) the transformation graph contains exactly 3 levels (transformation of inputs, modified sequence of instructions to instruction f_1, retransformation of the results).

Case 1: $s_{1,0}$ and $s_{1,1}$ are edges of TG_P'

$\Rightarrow trans(0) = trans(1) = 1$

$$\begin{aligned} E'(TG_P')^1 &= 0\cdot[...] + 1\cdot[0\cdot E(t_{2,0}) + E(s_{1,0}] + \\ &\quad + 0\cdot[...] + 1\cdot[0\cdot E(t_{2,1}) + E(s_{1,1})] \\ &= E(s_{1,0}) + E(s_{1,1}) \end{aligned}$$

Case 2: $s_{1,0}$ and $s_{2,1}$ are edges of TG_P'

$\Rightarrow trans(0) = 1, trans(1) = 0$

$$\begin{aligned} E'(TG_P')^1 &= 0\cdot[...] + 1\cdot[0 \cdot E(t_{2,0}) + E(s_{1,0}] + \\ &\quad + 0\cdot[...] + 1\cdot[1 \cdot E(t_{1,1}) + E(s_{2,1})] \\ &= E(s_{1,0}) + E(t_{1,1}) + E(s_{2,1}) \end{aligned}$$

<u>Case 3</u>: $s_{2,0}$ and $s_{2,1}$ are edges of TG_P'

$\Rightarrow$ trans(0) = trans(1) = 0

$$\begin{aligned} E'(TG_P')^1 &= 1\cdot[0 \cdot E(t_{1,0}) + E(s_{2,0}] + 0\cdot[...] + \\ &\quad 1\cdot[0 \cdot E(t_{1,1}) + E(s_{2,1})] + 0\cdot[...] + \\ &= E(s_{2,0}) + E(s_{2,1}) \end{aligned}$$

<u>Case 4</u>: $s_{2,0}$ and $s_{1,1}$ are edges of TG_P'

$\Rightarrow$ trans(0) = 0, trans(1) = 1

$$\begin{aligned} E'(TG_P')^1 &= 1\cdot[0 \cdot E(t_{1,0}) + E(s_{2,0}] + 0\cdot[...] + \\ &\quad + 0\cdot[...] + 1\cdot[1 \cdot E(t_{2,1}) + E(s_{1,1})] \\ &= E(s_{2,0}) + E(t_{2,1}) + E(s_{1,1}) \end{aligned}$$

$\Rightarrow$ With n=1 equation 3.1 is fulfilled.

<u>Induction Hypothesis</u>:

Let 3.1 be fulfilled for any $n \geq 1$, i.e. following equation holds:

$$E'(TG_P')^n = \sum_{i=0}^{n} \{(1-trans(i))\cdot[trans(i-1)\cdot E(t_{1,i}) + E(s_{2,i})] + trans(i)\cdot[[1-trans(i-1)]\cdot E(t_{2,i}) + E(s_{1,i})]\}$$

<u>Induction Step</u>:

For $E'(TG_P')^{n+1}$ follows with the induction hypothesis :

<u>Case 1</u>:

$s_{1,n}$ and $s_{1,n+1}$ are edges of TG_P' $\Rightarrow$ trans(n) = trans(n+1) = 1

The effort $E'(TG_P')^{n+1}$ ensues from the effort $E'(TG_P')^n$ that is required until the instruction f_n additional to the effort of the modified sequence $s_{1,n+1}$, because the instructions f_n and f_{n+1} are transformed with regard to the complement-on-one.

$\Rightarrow E'(TG_P')^{n+1} = E'(TG_P')^n + E(s_{1,n+1})$

Concerning 3.1 follows:

$E'(TG_P')^{n+1} = E'(TG_P')^n + 0\cdot[...] + 1\cdot[0 \cdot E(t_{2,n}) + E(s_{1,n+1})] = E'(TG_P')^n + E(s_{1,n+1})$

<u>Case 2</u>:

$s_{1,n}$ and $s_{2,n+1}$ are edges of TG_P' $\Rightarrow$ trans(n) = 1, trans(1) = 0

The effort $E'(TG_P')^{n+1}$ ensues from the effort $E'(TG_P')^n$ that is required until the instruction f_n additional to the effort of the transformation of the operands into the complement-on-two as well as to the effort of the modified sequence $s_{2,n+1}$, because the instructions f_n and f_{n+1} both are transformed with regard to different kinds of complement.

$\Rightarrow E'(TG_P')^{n+1} = E'(TG_P')^n + E(t_{1,n}) + E(s_{2,n+1})$

Concerning 3.1 follows:

$$E'(TG_P')^{n+1}$$
$$= E'(TG_P')^n + 1 \cdot [1 \cdot E(t_{1,n}) + E(s_{2,n+1})] + 0 \cdot [...]$$
$$= E'(TG_P')^n + E(t_{1,n}) + E(s_{2,n+1})$$

<u>Case 3</u>:

$s_{2,n}$ and $s_{2,n+1}$ are edges of TG_P' $\Rightarrow$ trans(n) = trans(n+1) = 0

The effort $E'(TG_P')^{n+1}$ ensues from the effort $E'(TG_P')^n$ that is required until the instruction f_n additional to the effort of the modified sequence $s_{2,n+1}$, because the instructions f_n and f_{n+1} both are transformed with regard to the complement-on-two.

$\Rightarrow E'(TG_P')^{n+1} = E'(TG_P')^n + E(s_{2,n+1})$

Concerning 3.1 follows:

$E'(TG_P')^{n+1} = E'(TG_P')^n + 1 \cdot [0 \cdot E(t_{1,n}) + E(s_{2,n+1})] + 0 \cdot [...] = E'(TG_P')^n + E(s_{2,n+1})$

<u>Case 4</u>:

$s_{2,n}$ and $s_{1,n+1}$ are edges of TG_P' $\Rightarrow$ trans(n) = 0, trans(1) = 1

The effort $E'(TG_P')^{n+1}$ ensues from the effort $E'(TG_P')^n$ that is required until the instruction f_n additional to the effort of the transformation of the operands into the complement-on-one as well as to the effort of the modified sequence $s_{1,n+1}$, because the instructions f_n and f_{n+1} both are transformed with regard to different types of complement.

$\Rightarrow E'(TG_P')^{n+1} = E'(TG_P')^n + E(t_{2,n}) + E(s_{1,n+1})$

Concerning 3.1 follows:

$$E'(TG_P')^{n+1}$$
$$= E'(TG_P')^n + 1 \cdot [1 \cdot E(t_{1,n}) + E(s_{2,n+1})] + 0 \cdot [...]$$
$$= E'(TG_P')^n + E(t_{2,n}) + E(s_{1,n+1})$$

Part 2 can be proven by analogy . ■

To obtain a practicable tool, the precompiler should generate the variant P_{min} that requires the minimal time effort. Let SV = $\{V_1, ..., V_m\}$ be the set of all variants that are represented by a transformation graph TG_P, then variant V_{min} complies with the following equation:

$$E(V_{min}) = \min \{E(V_1), ..., E(V_m)\}$$

Therefore, the problem that selects the most appropriate type of complement could be reduced to a problem of graph theory. If every edge of the transformation graph is weighted by the execution time that is required by the accompanying sequence of instructions, then the problem that has to be solved by the algorithms could be defined as: "Searching for the least cost path in the weighted graph TG_P". The determination of the least cost path is an often investigated problem, for which many algorithms are known [Nolt 76, Mehl 84].

4 Results

Based on the examinations of hardware fault coverage of divers applications made by [EHNi 90, Hinz 89], all faults that had not been detected by design diversity were analysed. For this examinations two diverse algorithms were used for different examples, which had been implemented in C. The diverse algorithms had used a set of library routines. As library routines correspond to a large number of instruction sequences which are executed identically in both variants, the used library routines were replaced by diverse routines. This replacement improved the detection of wrong results caused by a hardware fault, but it was not sufficient.
The detection of wrongly calculated results should be improved further by using systematically generated diversity additionally. To evaluate the achieved improvement of the fault coverage one of the design diverse variant was executed unchanged. The other variant was modified additionally by the implemented precompiler, before it was executed. Then the hardware fault detection was examined with the given software-implemented fault injector which injected the modelled faults while executing the different variants in the trace-mode.

The examination of the hardware fault coverage showed that calculating a wrong result could be perceptibly reduced while using systematically diversity additionally. This result is not astonishing, as still forced design diverse variants realize some functions often in a similar way (for example: transmission of input or output). Systematically generated diversity changes the use of the hardware and therefore reduces the possibility of the same falsification of the computed results.

5 References

EHNi 90 K. Echtle, B. Hinz, T. Nikolov: On Hardware fault detection by diverse software; 13th International conference on fault-tolerant systems and diagnostics, conf. proc., Verlag der Bulgarischen Akademie der Wissenschaften, 1990, S. 362 - 367.

HaGö 91 W. Hahn, M. Gössel: Pseudoduplication of floating point addition - a method of compiler generated checking of permanent hardware faults, Third European Workshop on Dependable Computing EWDC-3, Munich, April 1991.

Hinz 89 B. Hinz: Erkennung von Mikroprozessor-Hardware-Fehlern mittels diversitär entwickelter Software; Diplomarbeit, Fakultät für Informatik, Univ. Karlsruhe, 1989.

Mehl 84 K. Mehlhorn: Data Structures and Algorithms 2 - Graph Algorithms and NP-Completeness, Springer Verlag, 1984.

Nolt 76 H. Noltemeier: Graphentheorie mit Algorithmen und Anwendungen, de Gruyter, Berlin, 1976.

bezüglich = concerning, regarding
zugehörig = accompanying

recieved - erhalten (Vorgang)
obtained - erhalten (hinterher ist was da)