

Hardware Failure Virtualization Via Software Encoded Processing

Ute Wappler Christof Fetzer

Abstract—In future, the decreasing feature size will make it much more difficult to build reliable microprocessors. Economic pressure will most likely result in the reliability of microprocessors being tuned for the commodity market. Dedicated reliable hardware is very expensive and usually slower than commodity hardware. Thus, software implemented hardware fault tolerance (SIHFT) will become essential for building safe systems. Existing SIHFT approaches either are not broadly applicable or lack the ability to reliably deal with permanent hardware faults. In contrast, Forin [1] introduced the Vital Coded Microprocessor which reliably detects transient and permanent hardware failures, but is not applicable to arbitrary programs. It requires a dedicated development process and special hardware. We extend Forin’s Vital Code, so that it is applicable to arbitrary binary code which enables us to apply it to existing binaries or automatically during compile time. Furthermore, our approach does not require special purpose hardware.

I. INTRODUCTION

SOCIETY depends more and more on critical computing systems such as financial or x-by-wire systems. Safety related systems are typically built using special purpose hardware. However, such hardware is expensive because the number of units is typically much smaller than that of commodity systems. Also, such hardware is usually an order of magnitude slower than commodity hardware. Thus, we expect that in future there will be economic pressure to use commodity hardware for dependable computing. Therefore, one crucial step is the ability to transform value failures of hardware into crash failures—so-called fail-stop runs. The aim is that the probability that a value failure of the CPU is not transformed into a crash failure is negligible

The Vital Coded Processor (VCP) introduced in [1] performs a failure virtualization. It uses a combination of AN-codes and signatures to turn arbitrary compiler and hardware value failures into crash failures. AN-codes transform every data item to a multiple of A . Programs processing such data need to be adapted accordingly, e.g., branch conditions have to be changed. The output of AN-encoded programs is only valid if it is a multiple of A . For using the VCP, a special development process and special hardware is required. Furthermore, its applicability is restricted to programs whose dataflow is completely known at compile time. This paper aims at introducing the concepts required to make the principles of the VCP applicable to arbitrary binary code.

A valid question is, of course, if hardware failures are such an issue, that failure virtualization is required. Historically, hardware reliability has been increasing with every new generation. [2] states that in future the decreasing feature size of hardware will however not lead to more reliable but to

less reliable hardware—logic and memory. [3] impressively describes the effects of reduced feature sizes. Today’s CPUs have a variation in operating frequency of about 30% which is dealt with by using die binning, i.e., testing the resulting chips to find their operating frequency. This variability will increase further with decreasing feature sizes because of the following reasons [3]:

- **dopant variation** The threshold voltage of transistors is controlled by dopants inserted into the transistor channels. The smaller the transistors become, the less dopants are inserted. Thus, variations in this amount of dopants have a greater impact onto the electrical properties of the transistors. So, the uncontrollable variability of the production process leads to unpredictable properties of transistors.
- **subwavelength lithography** Nowadays, the wavelength of the light used in lithography is bigger than the produced structures. That makes the structures rough and also results in variations in the electrical properties of the produced transistors.
- **varying heat flux** How much heat is produced highly depends on the functionality of a building block and thus varies across the die. Since the transistor’s electrical properties are influenced by heat, transistors will have varying properties.

Increasing variability will make processor designs, at least as done today, more and more unpredictable. Apart from that, smaller features are less reliable because smaller transistors age faster and are more susceptible to soft errors since supply voltages decrease with decreasing feature size. It is expected that the amount of failures caused by soft errors increases exponentially with every new technology generation [3]. In future not only memory or logical building blocks under extreme conditions are sensitive to soft errors, but also hardware used under normal conditions.

II. RELATED WORK

Another, legitimate question is, why one should use the principles of VCP and not one of the many other existing solutions for detecting hardware failures.

On hardware level this problem is usually tackled by replication. [4] describes lock-stepped and loosely lock-stepped processors which make heavy use of space redundancy. That results in very expensive and complex hardware. An adjustment to programs with different safety requirements is not possible at runtime. Another, approach especially used in the avionics and aeronautics area is radiation hardening. This obviously only protects from soft errors. Permanent errors are not handled.

The hardware solutions which are able to handle errors in logic building blocks are very expensive. That typically

The authors are with the Technische Universität Dresden, Department of Computer Science.

Contact information: {ute.wappler, christof.fetzer}@inf.tu-dresden.de.

rules out the execution of non-safety-critical software on such hardware. Furthermore, they are not designed for detection of permanent design faults and have the disadvantage to require special hardware. On the other hand hardware solutions are mostly more efficient than software-based approaches. But since most processors are engineered for a commodity market, software-based techniques seem to be the most economic choice.

Control flow checking which can be implemented in hardware (e.g., [5]) or software (e.g., [6]) provides means to recognize invalid control flow for the executed program, that is execution of instructions which are not expected for the executed binary. Errors which do not influence the control flow will not be recognized by these approaches.

Existing software-based approaches do not provide as strong guarantees as VCP or are not applicable to general programs.

Algorithm-based fault tolerance [7] and self-checking software [8] use invariants contained in the executed program to check the validity of the generated results. This requires changes on the executed program and invariants have to exist. These invariants have to be designed to provide a good failure detection capability.

Various approaches using redundant executions exist: Simple time redundancy [9] cannot recognize deterministic permanent hardware faults and transient ones are also critical since they might influence the check comparing the results of redundant executions. The ED4I [10] approach also uses time redundancy but executes as a second version of the program a modified version which processes the same data as the original but encoded using an AN-code. The result of the transformed program has to be the A-multiple of the result of the original program. So, transient and many permanent hardware errors are recognizable. But faults in program loading, changed control flow, operand errors and permanent errors are not recognized if these influence both program versions. Also not recognized are errors in the code comparing the two results.

III. THE VITAL CODED PROCESSOR

The Vital Coded Processor (VCP) presented in [1] uses time, space and data redundancy to recognize transient and permanent errors. Programs executed by the VCP process data which is encoded using

- an AN-code to detect *data modifications* and faulty CPU operations (*operation errors*),
- variable dependent signatures to detect execution of wrong operations (*operator errors*) or usage of wrong operands (*operand errors*), and
- a time stamp to detect the usage of outdated operands (*lost updates*).

Instead of using the functional value x_f of the variable x the value is transformed to $x_c = A * x_f + B_x + d$. A is a variable independent constant. B_x is the signature of the variable x . The program is executed in a loop since VCP is applied to a programmable logic controller. d contains the number of already executed iterations (timestamp). The used code is an arithmetic code which means that it is preserved by arithmetic operations such as addition. For an addition of two encoded

variables x_c and y_c , it is possible to predict the signature of the result which will be $B_x + B_y + 2 * d$. In case of an addition, d will be subtracted to obtain a result encoded with a valid time stamp: $z_c = x_c + y_c - d = A * (x_f + y_f) + (B_x + B_y) + d$. If any of the above mentioned errors occur, z_c won't be a valid code word, that is, $(z_c - d) \bmod A$ will not result in the expected signature $B_x + B_y$.

Every input variable of a program is encoded using a signature which is chosen during the program development process and the current d provided by a hardware-implemented counter. The encoding is done by a special hardware before the input is given to the main CPU executing the encoded program. The signature of every dependent, i.e., by the program generated variable, can be precomputed offline using the source code of the executed program as the following example code demonstrates. Furthermore, it shows the corrective actions such as subtraction or addition of d which are required to obtain correctly encoded results.

```
// computing b+c+e encoded
int f(int b_c, int c_c, int e_c, int d){
    int t_c=b_c+c_c-d;
        // t_c = (A*b+Bb+d)+(A*c+Bc+d)-d
        // t_c = A(b+c)+Bb+Bc+d
    int a_c=t_c+e_c-d;
        // a_c = (A*t+Bt+d)+(A*e+Be+d)-d
        // a_c = A(b+c+e)+Bb+Bc+Be+d
    return a_c;
}
```

The signatures for input variables and precomputed signatures for output variables (for example $B_a = B_b + B_c + B_e$ in the previous listing) are stored in special purpose hardware. The latter ones are used by the hardware-implemented checker to test the validity of generated outputs (e.g., of a_c) by evaluating whether the following condition holds: $(a_c - d) \bmod A == B_a$. This check requires that all signatures are smaller than A , e.g., that $B_a < A$ holds. The functional value of a_c is obtained by integer division with A .

Control structures such as branches or loops are implemented in a way such that the signatures of all variables whose values depend on the control structure are independent of the chosen branch or the number of iterations. However, the signature will be incorrect if the wrong branch is taken or a wrong number of iterations are executed (see [1]).

The following pseudo code demonstrates the encoding of $\text{if}(x \geq 0) \{y = z + x\} \text{else} \{y = x - y\}$:

```
sigCond = sigGEZ(x_c);
if( x_c >= 0 ){
    y_c = add(z_c, x_c); // By=Bz+Bx
} else {
    y_c = sub(x_c, y_c); // By=Bx-By
    y_c = y_c - (Bx-By);
    y_c = y_c + (Bz+Bx); // By=Bz+Bx
    y_c = y_c - sigGEZ(<0) + sigGEZ(>=0);
}
y_c += sigCond; // By=Bz+Bx+sigGEZ(>=0)
```

$\text{sigGEZ}(x_c)$ computes the signature for the greater-equal-comparison of x_f with zero. It evaluates to two different values: one value if x_f is greater than or equal to zero and a different value if x_f is less than zero. The signature of y

after executing the if-statement is $B_z + B_x + sigGEZ(>= 0)$, independently of the chosen branch. If any of the computations or any of the used operands was faulty, the signature of y_c will be destroyed. The same happens if a branch is chosen which does not match x_f 's size.

To realize sigGEZ, signed numbers are represented using the two's complement as is done in most systems. When a negative value n_f and a positive value p_f are encoded using signed operations and the same signature h , the two encoded values $n_c = A * n_f + h$ and $p_c = A * n_c + h$ will not have the same signature if they are interpreted as unsigned values: $(p_f \bmod A) = h \neq (n_f \bmod A) = ((2^n + h) \bmod A)$.

An encoded while-loop would be encoded similarly. But in addition to checking the loop-condition, in the same way as the if-condition is checked, the number of executed iterations has to be checked using an additional iteration counter similar to the already used d . However, this additionally added iteration counter reduces the domain of the processed functional values. This issue is further aggravated when loops are nested.

VCP has the following disadvantages that restrict its use for execution of general software on commodity hardware:

- The complete data flow of the encoded program has to be known before the execution to be able to precompute the signatures of all output variables.
- The source code of all software components is needed.
- Encoding nested loops is a complex problem and reduces the domain of variables used in the loops.
- The signatures chosen for the input variables have to be selected in a way such that the signatures of output variables are smaller than A , or corrective actions have to be taken to ensure this property. The first option restricts the possible assignments—especially for large programs—whereas the second induces overhead during runtime.
- Special hardware is required to encode input variables, to store signatures, and to check the signatures of output variables.

IV. SOFTWARE ENCODED PROCESSOR (SEP)

The goal of the SEP is to provide the same safety as the VCP but without its disadvantages. SEP will execute arbitrary programs given as binaries on commodity hardware and turn hardware failures into fail-stop runs.

The main idea is to develop an interpreter which itself is encoded using the principles of VCP. This interpreter takes encoded input or encodes the input at runtime. Its input are the program to be executed and all input data processed by this program. The interpreter executes the program in an encoded fashion and thereby generates encoded output which is checked by another hardware unit, e.g., an onboard FPGA or a graphics processor.

We abstract from the dataflow of the executed program by associating a signature with each memory address of the virtual address space provided to the executed program. This signature does only depend on the memory *address* and a *version* which counts the number of memory updates. Thus, it is precomputable without knowing the dataflow of

the executed program. The whole process image, that is, processed data and program code, is encoded using the following code: $x_c = A * x_f + h(address, version)$. The signature $h(address, version)$ maps *address* and *version* to a number smaller than A . The dependence on *address* is required to recognize the usage of wrong operands, whereas the dependence on *version* is required to detect lost updates. The addition of the signature in general protects from unrecognized operator errors.

Our proof-of-concept implementation of the interpreter executes binaries which are compiled for the DLX-architecture—an academic instruction set architecture developed by Hennessy and Patterson [11]. It consists of operations handling the following instruction types: integer and floating point arithmetic, bit operations such as shifting or bitwise logical operations, comparisons, control flow instructions, and load/store operations. During the implementation we encountered several issues when applying the VCP-code to the DLX instruction set. These problems will have to be handled when applying the VCP-code to real-world architectures, too.

This paper mainly focuses on problems of encoding a CPU's instruction set. Details describing the encoding of the interpreter itself are out of scope.

V. PROBLEMS IN ENCODING BINARIES AND SOLUTIONS

A. Adherence to the CPU specification

The encoding solutions presented in [1] do not implement arithmetic operations with the same properties wrt. over- and underflows as in current CPU instruction sets. Since we are executing arbitrary binaries, we have to ensure that all instructions work as the compiler expects them to. Thus, it is required that the functional values, represented by the encoded numbers, are overflowing and underflowing as they would do without encoding.

In our current implementation, the encoded numbers are saved as 64-bit integers whereas the encoded functional values are 32-bit values. In general, it will be always the case that encoded numbers have a bigger domain than unencoded ones. Both are wrt. addition and multiplication algebraic structures of the type ring. But they are not isomorphic, because A must not be equal to 2^{32} . Choosing A to be a power of two would result in a hamming distance of one between valid code words—rendering the code pretty useless against a wide variety of bitflips. Since encoded and functional values are non-isomorphic, encoded numbers will not overflow or underflow, in cases where the encoded functional 32-bit values would do so. This means that for each operation, we have to check if there would have been an underflow or overflow within the functional values and if so, correct the obtained encoded result accordingly.

The type and amount of corrective actions depend on the kind of encoding which is used. We could either interpret the encoded numbers as signed numbers (usually stored using the two's complement) and use signed operations to encode them (*signed encoding*) or interpret them as unsigned and use unsigned operations (*unsigned encoding*). It is possible to transform both representations into each other with slightly

increasing the probability for unrecognized errors. Signed encoded numbers are required for realization of encoded comparisons. But operations using unsigned encoded numbers usually require less corrective actions to be taken. For a performance comparison see section V-C.

Figure 1 demonstrates the addition of two signed encoded values: $s1_c = A * s1_f + h1$ and $s2_c = A * s2_f + h2$. It can be seen that the sum of the encoded values $s1_c + s2_c = A * s1_f + h1 + A * s2_f + h2$ is not equal to the value obtained when encoding the sum of the functional values: $A * (s1_f + s2_f) + h1 + h2$. In that case, a correction by `OVERFLOW_COR` is required.

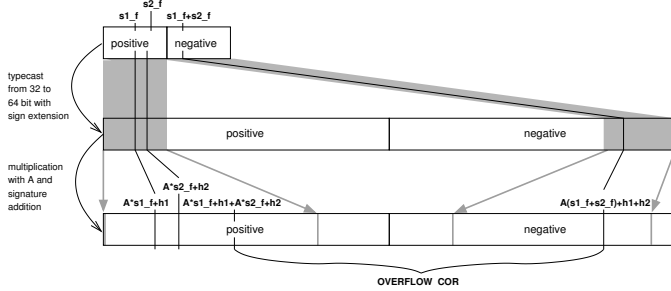


Fig. 1. Signed addition of positive numbers overflowing to negative.

Similar correctional steps are required if two negative numbers are added and the result is overflowing to the positive numbers. The following listing shows the corrective actions required for a signed addition of signed encoded values $s1_c$ and $s2_c$.

```
// return the functional value of v
extern uint32 getFunctionalValue(uint64 v);
const uint FIRST_NEG = 2^31;
const uint OVERFLOW_COR = 2^64 - A * 2^32;

uint64 add(uint64 s1_c, uint64 s2_c){
    // Addition of encoded values
    uint64 result = s1_c + s2_c;

    // Corrections
    uint32 s1_f = getFunctionalValue(s1_c);
    uint32 s2_f = getFunctionalValue(s2_c);
    // Did an overflow from
    // positive to negative occur?
    if (s1_f < FIRST_NEG && s2_f < FIRST_NEG
        && s1_f + s2_f >= FIRST_NEG){
        result += OVERFLOW_COR;
    }
    // Did an overflow from
    // negative to positive occur?
    if (s1_f >= FIRST_NEG && s2_f >= FIRST_NEG
        && s1_f + s2_f < FIRST_NEG){
        result -= OVERFLOW_COR;
    }
    return result;
}
```

Most other operations such as signed subtraction, multiplication etc. do require similar corrections whose amount depends on the chosen encoding scheme (signed or unsigned). These corrections are a problem. Soft errors might cause them to be left out. That will result in a valid and decodable result.

But the functional value in the 64-bit domain, i.e., before type-casting back to 32-bit during decoding, will contain invalid sign bits. That will cause unrecognized errors when it is used in comparisons or signed multiplications. This issue arises with all corrective action.

B. Encoding Instructions

Forin [1] presents the realization of addition, subtraction, comparison with zero, if- and while statement. He leaves open how to encode other relevant operations such as multiplication or division and their application on signed and unsigned data types. Of course, an implementation of these operations using the operations presented by Forin is possible, but surely will be less efficient than a directly encoded implementation using the multiplication or division operations provided by the processor.

a) *Multiplication*: When multiplying two encoded numbers $m1_c$ and $m2_c$, several corrective actions are required to obtain a valid encoded result with a signature which does only depend on the signatures of $m1_c$ and $m2_c$. The following listing demonstrates one possible implementation¹. The signatures of $m1_c$ and $m2_c$ are denoted as $h1$ and $h2$. $m1_f$ and $m2_f$ denote the functional values of $m1_c$ and $m2_c$. A is assumed to be a global constant requiring at most 31 bits.

```
uint64 mult(uint64 m1_c, uint64 m2_c){
    uint128 result = (uint128)m1_c * (uint128)m2_c;
    // result = A^2 * m1_f * m2_f
    //           + A * m1_f * h2 + A * m2_f * h1
    //           + h1 * h2
    uint128 tmp1 = (result / A % A) * A;
    // tmp1 = A * m1_f * h2 + A * m2_f * h1
    uint64 tmp2 = result % A;
    // tmp2 = h1 * h2
    result = (result - tmp1) / A;
    // result = A * m1_f * m2_f
    result += tmp2;
    // result = A * m1_f * m2_f + h1 * h2
    return (uint64)result;
}
```

This solution poses the following severe restrictions onto the encodable functional values and usable signatures:

- $m1_f * h2 + m2_f * h1 < A$
- $h1 * h2 < A$

A further issue is that before the last step, the result is available as a multiple of A but without any signature— $A * m1_f * m2_f$. There is the very small but existent probability that at that point, the value might become mixed up with other multiples of A stored in memory or in registers.

Obviously, these restrictions make this implementation practically unusable. Thus, we developed a second version, which requires knowledge of the used signatures and functional values but does not have these issues. The functional values can be obtained from the encoded values using integer division. The signatures $h1$ and $h2$ are known beforehand in the VCP approach or are precomputable at runtime in our SEP interpreter:

¹For an improved comprehensibility the following listings are simplified, e.g., without overflow corrections.

```

uint64 mult(uint64 m1_c, uint32 h1,
            uint64 m2_c, uint32 h2){

    uint32 m1_f, m2_f;
    m1_f=getFunctionalValue(m1_c);
    m2_f=getFunctionalValue(m2_c);

    uint128 result=(uint128)m1_c*(uint128)m2_c;
    // result=A^2*m1_f*m2_f
    //      +A*m1_f*h2+A*m2_f*h1
    //      +h1*h2
    result -=A*(uint128)(m1_f*h2+m2_f*h1);
    // result=A^2*m1_f*m2_f
    //      +h1*h2
    result +=(A-1)*(uint128)(h1*h2)
    // result=A^2*m1_f*m2_f
    //      +A*h1*h2
    result /=A;
    // result=A*m1_f*m2_f+h1*h2
    return (uint64)result;
}

```

The multiplication will require overflow correction similar to the addition. That is problematic since soft errors might lead to its omission.

b) Division: An encoded division operation can be implemented using an encoded while-loop, encoded addition and subtraction. But in that case, the processor's division implementation, which is surely faster than a software solution, would not be used. The following implementation is more efficient:

```

uint64 div(uint64 d1_c, uint32 h1,
           uint64 d2_c, uint32 h2){

    uint64 result=(A*(d1_c-h1))/(d2_c-h2);
    result +=(d1_c%A)/(d2_c%A);

    return result;
}

```

This implementation bears the risk of unrecognized operand errors. The signature of the operands is removed in the first step. At that point the operands might be exchanged with other multiples of A which might be stored in a register used by mistake. The multiplication with A has to be done before the division. Otherwise, the result would be completely unencoded for a short time. For an application with very high safety requirements the division should be implemented using encoded addition, multiplication, if and while. That results in the already stated drawbacks of reducing the domain of processed data and large computational overheads.

c) Comparisons: All comparisons can be reduced to comparisons with zero. A prerequisite to an encoded comparison with zero, not mentioned in [1], is that encoded value and functional value have to have the same sign when interpreted as signed numbers. That means encoding has to be done using signed operations.

d) Logical operations: Even if logical operations are not part of the executed instruction set, they are useful to implement conditional branches or comparisons. If boolean values are stored as one and zero, logical operations can be

easily implemented using arithmetic operations:

$$\begin{aligned}
 a||b &= a + b - a * b \\
 a\&\&b &= a * b \\
 !a &= 1 - a
 \end{aligned}$$

e) Shift operations: Shift operations can be emulated using division for right shifting or multiplication for left shifting. Obviously, that makes them slower than a compiler might expect them to be.

f) Control flow operations: Control flow operations change the Program Counter (PC). To be able to check that these changes are executed correctly, we have to encode the PC and do all computations changing it in an encoded fashion. The PC is encoded using the following formula: $PC_c = A * PC_f + h(PC_f, version)$. The reference to its own functional value within the signature enables us to check if executed instructions match the current PC. Remember, instructions are encoded in the same way as data is: $inst_c = A * inst_f + h(address, version)$. Because the PC contains the address from which an instruction is loaded the signature $h(PC_f, version)$ of the PC and the signature $h(address, version)$ of the instruction have to be equal. The sum of both is added to every data item modified by an executed instruction. If the executed instruction would not match the current PC, the code of the generated data would be invalid.

g) Load/store operations: Load/store operations are a simple move. They just require adaptation of the signature of the moved value to match the new address by adding the new signature and subtracting the old one.

h) Further instructions: Currently, we cannot encode bitwise logical operations and floating point instructions natively. Thus, they will have to be encoded by emulation using the encoded operations presented beforehand.

C. Signed vs. Unsigned Encoding

Table I compares the slow down induced for the different forms of encoding. Obviously, operations on unsigned encoded numbers mostly induce less overhead than the ones using signed encoded values. That was to be expected, because the latter ones require more corrections to ensure adherence to the CPU's specification. Even for operations which require the transformation from unsigned to signed encoded values (signed division and greater then comparison), the induced overhead is not much larger for the unsigned encoded variant. That again can be explained by the overhead generated by corrective actions.

The equals comparisons is especially slow since it is emulated by computing: $!(a > b || a < b)$. That requires two comparisons, an addition, a subtraction, and an expensive multiplication.

VI. FAULT INJECTION EXPERIMENTS

We have implemented a SEP interpreter using the principles presented in this paper. Currently, we are able to execute simple C-programs compiled to the DLX-architecture which

| Operation using values: | encoded signed | encoded unsigned |
|-------------------------|----------------|------------------|
| unsigned addition | 24 | 3 |
| signed subtraction | 12 | 2 |
| signed multiplication | 52 | 38 |
| signed division | 18 | 15 |
| signed greater then | 27 | 30 |
| equals | 169 | 125 |

TABLE I

SLOW DOWN IN X TIMES SLOWER THAN NATIVE EXECUTION.

can make use of nearly the whole DLX instruction set, apart from floating point instructions and bitwise logical operations. We also provide a simple set of system calls which implement basic I/O-functionality such as `printf` and file operations.

To test the fault detection capabilities of SEP, we used our simulation-based hardware fault injection tool FITgrind [12]. FITgrind was used to probabilistically inject errors of the following types into the running interpreter or native execution: (1) bitflips in memory and operation results and (2) replacement of instructions. The results of an injection run were compared with an error free (golden) run. We checked if any erroneous output, i.e., output differing from the golden run, was generated and if an error was recognized either by the operating system or the SEP interpreter.

For testing, we used a program computing the MD5 hash of a string which we executed around 8000 times for each variant. For each run, FITgrind was initialized with another random number and thus generated different error patterns. The results can be classified as follows:

- **Correct and complete** The generated output did not differ from the golden run, despite the injected errors.
- **Correct but incomplete** The generated output was not complete but a prefix of the output of the golden run.
- **No output** No output was generated, that is, the application crashed/stopped before generating any output.
- **Incorrect output** Output was generated that differed from the golden run.

The first three output types are safe, i.e., no erroneous output is generated. But the last type—incorrect output—represents an unsafe behavior. If a system generates this type of output, it is not fail-stop. While the SEP interpreter produced no unsafe run at all, 9% of the native executions did so. On the other hand, 31% of all native runs produced complete and correct output despite the injected errors. For the encoded interpreter, only 0.06% of the runs under fault injection produced completely correct results. That shows that the interpreter detects errors before they can propagate and even errors which would not become visible. This can be used as a warning that a system might break down in future.

87% of the errors detected for the interpreter were detected by the operating system. Only a small part (8%) was detected by checking if the output is a valid code word. The interpreter itself does consistency checks apart from code checking which lead to further detections (5%). Every case of failure (*correct but incomplete* or *no output*) was indicated by an exception or error signal. For the native execution 14% of the failed runs produced no error indication, i.e., these failures were only recognized because the outputs of golden run and injected run differed.

| output type: | encoded | native |
|------------------------|---------|--------|
| Correct and complete | 0.06% | 30.86% |
| Correct but incomplete | 17.26% | 0% |
| No output | 82.68% | 60.11% |
| Incorrect output | 0% | 9.03% |

TABLE II

OUTPUT TYPES OF FAULT INJECTION EXPERIMENTS IN % OF RUNS.

VII. CONCLUSION

We proved that encoding binaries using Forin's [1] encoding principles is possible. In our fault injection experiments no violation of the desired fail-stop behaviour occurred, despite the mentioned problems in encoding operations such as multiplication and division and the inability to ensure the execution of corrective actions. Applying the VCP encoding principles to binaries enables us to encode arbitrary programs without knowing their data flow beforehand.

Further research is required to investigate the probability of undetected failures. Also, the performance has to be improved. Thus, our current work aims at pre-encoding and compiling as much code as possible using the principles of Forin but on assembler level instead of source code level. Interpretation shall only be done for indirect, i.e., not predictable, control flow transfers. That will require an adapted compilation process for safety-relevant software but will provide crash failure virtualization for nearly arbitrary hardware failures.

REFERENCES

- [1] P. Forin, "Vital coded microprocessor principles and application for various transit systems," in *IFA-GCCT*, Sept 1989, pp. 79–84.
- [2] L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective," *IBM Journal of Research*, vol. 43, 1999.
- [3] 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.
- [4] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," in *DSN 2005*. Los Alamitos, CA, USA: IEEE Computer Society, 2005.
- [5] X. Li and J.-L. Gaudiot, "A compiler-assisted on-chip assigned-signature control flow checking," in *ACSAC 2004*, 2004, pp. 554–567.
- [6] S. Bagchi, Z. Kalbarczyk, R. Iyer, and Y. Levendel, "Design and evaluation of preemptive control signature (PECOS) checking," *IEEE Transactions on Computers*, 2003.
- [7] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, vol. 33, no. 6, 1984.
- [8] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *J. ACM*, vol. 44, no. 6, pp. 826–849, 1997.
- [9] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," in *DATE*. IEEE Computer Society, 2003, pp. 20 057–20 063.
- [10] N. Oh, S. Mitra, and E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, 2002.
- [11] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*. CA, USA: Morgan Kaufmann Publishers Inc., 1990.
- [12] U. Wappler and C. Fetzer, "Hardware fault injection using dynamic binary instrumentation: FITgrind," in *Sixth European Dependable Computing Conference (EDCC-6)*, vol. Proceedings Supplemental Volume, October 2006, pp. 37–38.

Ute Wappler received her diploma in 07/2004. Since then she is a PhD student at the Dresden University of Technology, Department of Computer Science (Systems Engineering Group). Her research interests are software implemented hardware failure virtualization and hardware fault injection.

Christof Fetzer received his Ph.D. from UC San Diego (1997). He joined AT&T Labs-Research in 1999. Since April 2004 he has an endowed chair (Heinz-Nixdorf endowment) in Systems Engineering at the Dresden University of Technology.