Abstract

We expect that in future commodity hardware will be used in safety critical applications. But the used commodity microprocessors will become less reliable because of decreasing feature size and reduced power supply. Thus software-implemented approaches to deal with unreliable hardware will be required. As one basic step to software-implemented hardware-fault tolerance (SIHFT) we aim at providing failure virtualization by turning arbitrary value failures caused by erroneous execution into crash failures which are easier to handle. Existing SIHFT approaches either are not broadly applicable or lack the ability to reliably deal with permanent hardware faults. In contrast, Forin [7] introduced the Vital Coded Microprocessor which reliably detects transient and permanent hardware errors but is not applicable to arbitrary programs and requires special hardware. We discuss different approaches to generalize Forin’s approach and make it applicable to modern infrastructures.

1. Introduction

Nowadays, safety related systems are typically built using special purpose hardware. The used hardware is either radiation hardened to prevent environment induced execution errors or provides hardware redundancy to detect and correct such errors. However, these solutions are expensive because the effort put into design and production is much higher and the number of units is much smaller than for commodity systems [2].

We expect that in future there will be economic pressure to use commodity hardware for dependable computing. Furthermore, systems will become mixed mode, i.e. facilitate the execution of both critical and non-critical applications on the same computer system built from unreliable hardware. Historically, hardware reliability has been increasing with every new generation. But in future, the decreasing feature size of hardware will not lead to more reliable but to less reliable hardware. Borkar in [4] impressively describes the effects of reduced feature sizes. The conclusion is that the uncontrollable variety of the production process will make processor designs more and more unpredictable. Furthermore, smaller transistors age faster and thus become faster unreliable and smaller features 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.

Thus mixed mode systems will require new dependability mechanisms which make it possible to cope with the restrictive failure detection capabilities of commodity hardware. One crucial step in providing such mechanisms is failure virtualization, i.e. the transformation of a (more difficult to handle) failure model into another (easier to handle) failure model.

One approach implementing such a failure virtualization is the Vital Coded Processor (VCP) by Forin [7] which uses arithmetic codes to detect runtime errors and can provide a negligible probability of non-detection of permanent and transient errors. But the applicability of VCP is restricted to small programs and a dedicated hardware environment. The objective of our research is to make the techniques of VCP applicable to a broad range of programs using commodity hardware. In this paper, we will discuss different ways of achieving our objective and compare them to the requirements posed by industry. We will not present a complete solution to the problems which we will point out.

2. Related Work

A widespread technique to detect hardware errors in memory are error correcting codes (ECC) and parities which are used to detect and correct inconsistent data. But soft errors do not only corrupt memory but can also influence the logic building blocks. The resulting errors are not detectable using ECC. Furthermore, usually only large memory banks and not every register is protected using ECC because the latter would be way too much effort.

Control flow checking, e.g. [10, 3, 1], provides means to recognize invalid control flow, that is execution of in-
structions which are not expected for the executed binary. This will detect mostly errors resulting in an erroneous program counter. If processed data is modified or a single non-control flow instruction such as an addition is executed erroneously, that is not detectable using only control flow checking. Control Flow checking approaches do also not provide adjustable guarantees.

Algorithm based fault tolerance (ABFT) [15] uses invariants contained in the executed program to check the validity of the generated results, thereby checking validity of control flow and data. The provided error detection capabilities depend on the available invariants. Thus, ABFT has only a very constrained applicability.

Various approaches use redundant execution on software level to detect hardware errors. E. g. [16, 13, 12] execute instructions multiple times and compare their outcome to detect and correct errors. This redundancy additionally can be combined with control flow checking [14]. These approaches are not able to detect design faults and permanent errors in the used hardware. The solutions described in [11, 5, 8, 6] execute automatically generated different program versions and compare the obtained results. But still the detection of permanent hardware errors cannot be guaranteed.

3. Vital Coded Processor (VCP)

The Vital Coded Processor (VCP) presented in [7] uses time, space and data redundancy to recognize transient and permanent errors disturbing program execution. Therefore, data as well as the executed program are modified. Programs executed by the VCP process data which is encoded using an AN code with signatures and timestamps. For every variable \( x \), instead of its original functional value \( x_f \) its encoded version \( x_c = A \cdot x_f + B_x + D \) is used in program execution. \( A \) is a variable independent constant and multiplication with \( A \) forms the AN code. \( B_x \) is the signature of the variable \( x \) and assigned when the program is encoded, i.e. transformed to process encoded variables instead of unencoded ones. With the VCP every program is executed in a loop since the VCP uses a programmable logic controller. \( D \) contains the number of already executed iterations (timestamp). With every iteration the encoder takes new input encodes it using the next \( D \). See figure 1 for a VCP example executing the program \( \text{return } z = x + y \).

This code ensures the detection of the following errors:

- the AN-code detects operand modifications and faulty CPU operations (operation errors),
- the signatures \( B_x \) facilitate detection of the execution of wrong operations (operator errors) or usage of wrong operands (operand errors), and
- the time stamp \( D \) detects the usage of outdated operands (lost updates).

![Figure 1. VCP environment.](image1)

Every input variable of a program—\( x \) and \( y \) for our addition example—is encoded using its signature \((B_x \) and \( B_y)\) which is chosen during the program development process and the current \( D \), provided by a hardware-implemented counter. The encoded 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 and the signatures of the input variables. For our example the signature of \( z_c \) without corrections would be \( B_x + B_y + 2 \cdot D \). To obtain a code word with a valid timestamp, \( D \) has to be subtracted once resulting in an encoded program: \( z_c = x_c + y_c - D \) which on runtime computes: \( A \cdot (x_f + y_f) + (B_x + B_y) + D \).

The signatures for input variables and precomputed signatures for output variables are stored in special purpose hardware. The latter ones are used by the hardware-implemented checker and decoder to test the validity of generated outputs by evaluating whether the signature contained in the computed data equals the precomputed signature, that is if \((z_c - D) \mod A == B_x + B_y \) holds. If any of the above mentioned errors occurs the equation most probably will not hold, i.e. \( z_c \) is no valid code word. In this case

![Figure 2. VCP workflow.](image2)
the checker will stop the CPU and turn all output pins into a safe state.

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 [7] for details). This requires additional check values which are added to the code words. For example for each nested loop another iteration counter \( D_3 \) has to be added to the affected code words.

Figure 2 depicts the workflow required to produce an encoded program and a list of signatures for input variables and output variables. First, unencoded source code is encoded, that is the data types of all variables are adapted to match the domain of encoded values and all operators and control structures are replaced by their encoded counterparts. Furthermore, the encoding transformation computes the signatures for input and output values. After compilation of the resulting encoded program, it has to be linked to the library of encoded operators. As a result an encoded binary is obtained.

VCP has the following disadvantages that restrict its use for execution of general software on commodity hardware: First, 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. That excludes the usage of dynamically allocated memory and function pointers and thus prevents event driven programming. Second, special hardware is required to encode input variables, to store signatures, and to check the signatures of output variables. Last, encoding of nested control flow statements is a rather complex task because several timestamps and check values have to be considered. It might even reduce the safety which can be guaranteed since overflows of the code words must be prevented which for example will require to have upper bounds for loop iteration counters, that is these counters will be set back to zero when reaching this bound. Another option is to restrict the number of possible iterations.

But on the other hand VCP provides a predictable and very high error coverage and is compliant to safety standards such as IEC 61508. Apart from VCP no codes with signatures were used so far. If considered at all, the problem of wrong operand, operator, and lost update errors was handled by parallel execution of multiple versions of a program. In that case the applied error model is less restrictive. There might exist permanent faults modifying address lines which influence both program versions and thus are not detectable. These are admittedly rare but cannot be excluded.

Furthermore, the achievable degree of safety can be influenced by the choice of \( A \). The probability that a number of bitflips modifying a codeword, which encodes a functional value of the maximum size of \( n \) bits, remains undetected is approximately:

\[
\frac{\text{number of valid code words} - 1}{\text{number of possible words}} \approx \frac{2^n}{2^{2n/\log_2 A}} = \frac{1}{A}.
\]

Furthermore, VCP also detects errors introduced by the compiler or linker since encoding is done on the source code level.

Our objective is to remove existing restrictions of VCP and make it applicable to modern mixed-mode computing systems. Therefore, we have to ensure that it is applicable to a broad range of programs (completeness), is easy to use (usability) and its safety can be quantified (safety assessment).

**Completeness** To ensure its broad applicability, we have to provide encoded counterparts for as many operators and programming constructs as possible. We especially aim at providing support for dynamically allocated memory and statically not known control flow—both are currently not supported by VCP. Furthermore, industry requests for example encoded versions of trigonometric functions.

**Usability** The developer of an encoded application should have to know nothing about encoding apart from 1) the restricted operator set which can be used and 2) the massive overhead induced by encoding. Thus we have to provide appropriate development tools which automate encoding and signature assignment.

For example, natively encoded floating point numbers are currently not supported. If encoding is realized with the help of an encoding compiler which encodes code provided by the developer, the usage of not available operators will result in compiler errors which might be confusing. On the other hand, a graphical programming tool which restricts the developer to the available encoded operators makes handling of legacy code more difficult. But it also cannot be expected that an encoding compiler could handle legacy code without producing errors because of the unavailability of encoded operators. Furthermore, support for debugging of encoded applications will be required.

Encoding requires not only support for larger data types but is also very time consuming. For example an encoded addition takes three times as long as an unencoded one. For multiplication we even encounter a slow down factor of 38 [17] and have to be able to process values which are twice as large as the code words themselves. Thus, the developer should only encode absolutely safety critical parts of a system especially in the face of the very restricted resources within most safety critical systems, e. g. in the automotive industry. For example, consistency checks could be encoded while the main program remains unencoded. Less critical system parts could be encoded using less safe and less performance intensive codes, e. g. AN codes without signatures.

**Safety Assessment** The so far presented theoretical safety assessments of VCP assume a constant Hamming Distance between all code words and the error model is that...
of equally distributed bit flips modifying code words during storage. The assumptions do not reflect reality where single bit flips are more probable than multiple bit flips and the Hamming Distance between the code words of an AN code with signatures is not equally distributed as we have seen in experiments. Additionally, not the whole supported error model is covered. For example bit flips on address lines, the instruction pointer or in the code section are completely ignored. But a practical safety assessment using error injection can mitigate the inability to define a sufficient system model for safety assessment and increases trust with a growing number of experiments.

In the following we will concentrate on completeness of control flow and memory management and the usability of encoding.

4. Encoding Variants

Arithmetic encoding in software can be done on different levels of abstraction: on the source code as in VCP or on assembler code either on compile or on runtime. The decision on which level encoding is done influences how the workflow used to develop encoded applications is modified, and how large the provided sphere of protection is.

Encoding on assembler level—both on compile and run time—provides a smaller sphere of protection since encoding is done within or after the compilation and linking. Thus, compiler and linker are partly or completely respectively unchecked. On the other hand encoding on assembler level on runtime does not change the required workflow. For assembler level encoding on compile time we need to replace the used compiler with an encoding one. That surely is a problem in productive embedded systems where it is not possible to replace compilers. The option is to use a compiler framework such as the LLVM framework [9] which can provide different backends, e. g. one which produces C code which can be used as input to the unreplacable compiler.

Furthermore, we hope that control flow might be easier to encode on assembler level than on source code level since the available control flow operators are much less complex than in any programming language. On assembler level we will never encounter nested control flow structures and it is not required to differentiate between loops and branches. We can directly encode control flow by for example encoding the instruction pointer used for execution. This explicit encoding of control flow allows for an easy handling of function pointers and thus facilitates event driven programming. On the other hand we obviously loose information about loops which might make lost update protection more difficult.

Supporting dynamically allocated memory does not depend on the level of encoding. It can be implemented by using address dependent signatures as introduced in [17] and specific encoded load and store operations which will result in an invalid code word if a value is read from or stored to the wrong address.

Software Encoded Processing (SEP) Software Encoded Processing (SEP) which we introduced in [17] implements encoding on assembler level on runtime. The main idea was to develop an interpreter for programs given as binary. This SEP interpreter itself is encoded using the principles of VCP [7], that is every variable which is crucial to the correct execution of a program is encoded. This includes the program executed by the interpreter and data processed by that program, but also data used by the interpreter to manage program execution such as the instruction counter. Since we do not know beforehand how much memory is used by a process image, that is code and data of the executed binary, the interpreter uses signatures which are not statically assigned but depend on the address of the code word. Since we do not statically know which data item will be changed when, the signature does also depend on the number of executed instructions (instruction counter)—the so-called version of a data item. This version is required for lost update detection and is equivalent to the D and the loop iteration counters of VCP. But we have to manage an additional (also encoded) data structure which gives us the version information to an address. Otherwise, we could not decode the obtained results correctly.

Thus, the SEP interpreter executes the program in an encoded fashion and thereby generates encoded outputs which are checked by another (standard) hardware unit, e. g. an FPGA which implements the checker whose task it is to determine if the generated output consists of valid code words. Therefor, the checker has to be able to compute the expected signature which requires knowledge of the address of the checked code word and of the expected version. See figure 3 for an overview of the system.

Checker and interpreter execution, both might be influenced by transient errors. Because of that, additional hardware used for code checking should be protected from unrecognized transient errors. Either hardware redundancy is used in the form of diverse code checkers connected in a redundancy scheme such as triple or double modular redundancy. Or software redundancy is used, e. g. repeated execution and comparison or usage of arithmetic codes.

The advantage of SEP is that every binary can be executed encoded without requiring its source code. Furthermore, any kind of control flow as well as dynamically allocated memory are supported. No explicit encoding for function calls—direct or via function pointer—for if-statements or loops is required. On the other hand, as already stated the sphere of protection is reduced. Protection only starts when the process image either is encoded or a signature of it is
computed which facilitates encoded checking for modifications. The greatest disadvantage is the overhead generated by interpretation. Encoded operations are already significantly slower than unencoded ones. Interpretation and signature management further worsen performance.

Considering the usability, no modification of the workflow is required but if a programmer uses unsupported operators such as floating point computations this will result in a runtime error. Thus, it will be noticed pretty late in the development process or not at all if test coverage is not 100% and the unsupported operator usage occurs in an untested part of the code.

**Compiler Encoded Processing (CEP)**  The intention of CEP is to implement encoding on assembler level but to do the encoding on compile time for example using the LLVM compiler framework [9] and its intermediate code which is a single assignment assembler-like language. This enables us to implement explicitly encoded control flow similar to SEP. Thus, we will be able to encode control flow which is not known on compile time. Furthermore, we can assign variable dependent signatures as used in VCP to every variable which is already known on compile time and dynamic address and version number dependent signatures as used in SEP to dynamically allocated memory. That facilitates support for dynamically allocated memory and faster and safer to encode statically known variables.

On runtime no interpreter is required but as with SEP and VCP an external checker. See figure 4 for an overview of the system. The checker does not check if the sent check values are valid code words. CEP will have an error collecting variable whose value is completely statically precomputed and formed by all executed computations. The checker will have a watchdog functionality, that is it will check if regularly a check value arrives and if this value is equal to its expected statically on compile time precomputed value.

In comparison to SEP CEP requires a significantly changed workflow which is similar to VCP’s. First, the compiler frontend transforms the code into the intermediate assembler-like language (byte code) used for encoding which is done by the next step, the encoding pass. The resulting encoded byte code is compiled and linked with libraries containing implementations of encoded operators and encoded memory which implements encoded load and store operations which handle address and version dependent signatures. The encoding step does also produce the check values used by the checker to detect execution errors.

**5. Example Application: Quadcopter**

As an example application to test practicality of our architecture a remote controlled helicopter with four rotors will be used (see figure 6). Sensor data which first passes a noise filter is processed by the flight control software. Based on a physical model of the quadcopter the control loop calculates the motor control signals. The monitoring channel checks the vital flight data, such as inclination, acceleration and motor speed. When the quadcopter leaves the safe flight envelope the monitor takes an emergency action, e.g. switching off the engines and releasing a parachute. The monitor is the safety critical part of the system which shall be encoded. The checker will be implemented using
a small ATMEL 8-bit controller. Its synchronization with the monitoring channel on the TriCore is crucial. Furthermore, the checker has an error counter. The system is only brought to its fail-safe state if the error counter exceeds a certain threshold. For testing the checker, the monitor will inject erroneous check values. In case of an erroneous check value the ATMEL increases its error counter whose value is sent to the monitor. A valid check value reduces the error counter. Thus, the monitor can check the validity of the checker and can also bring the system to its fail-safe state in case of an invalid checker. So, monitor and checker validate each other.

Since the signal filtering has to be encoded in the monitor and various calculations are necessary, the monitor contains non trivial software. The case study is also meant to identify missing encoded operations.

6. Conclusion

We have pointed out that in future hardware will become more susceptible to transient errors and because of its growing complexity also to permanent errors. Thus, it can no longer be assumed that software is executed correctly. Furthermore, dedicated safe hardware which is more reliable is very expensive and slower than its non-reliable counterparts. Software implemented hardware fault-tolerance is a flexible and less expensive option to prevent silent data corruptions caused by hardware errors because it enables us to turn these erroneous output values into system crashes which are easier to handle. Encoded processing as presented by Forin (VCP) provides a very high safety standard but is not generally applicable since it requires a very specific system environment. We pointed out the disadvantages and identified possible solutions.

References


