

Software Encoded Processing: Building Dependable Systems with Commodity Hardware

Ute Wappler and Christof Fetzer

Technische Universität Dresden
Department of Computer Science
<http://wwwse.inf.tu-dresden.de>
Dresden, Germany
{ute.wappler,christof.fetzer}@inf.tu-dresden.de

Abstract. In future, the decreasing feature size and the reduced power supply 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. In the dependability domain we expect the continued spreading of mixed-mode computing systems, i.e., systems that execute both critical and non-critical functionality. To permit the efficient execution of non-critical applications and the correct execution of critical applications, we introduce the concept of Software Encoded Processing (SEP). SEP enforces a crash failure semantics of the underlying CPU. It does not require the source code of encoded programs and provides probabilistic guarantees. To achieve this, arithmetic codes and signatures are used to detect corrupted data and faulty executions of programs.

1 Introduction

Society depends more and more on critical computing systems such as financial systems 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 much smaller than that of commodity systems. Also, such hardware is usually an order of magnitude slower than commodity hardware. We expect that in future there will be economic pressure to use commodity hardware for dependable computing. Systems will need to facilitate the execution of both critical and non-critical applications on the same computer system. Such mixed-mode commodity 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 the ability to transform value failures of commodity hardware into crash failures: the aim is that the probability that a value failure of the CPU is not transformed into a crash failure is negligible. The concept of Software Encoded Processing (SEP) presented in this paper guarantees such a *failure virtualization*, i.e., the transformation of a (more difficult to handle) failure model into another (easier to handle) failure model.

A valid question is, of course, if such a SEP is indeed needed. Historically, hardware reliability has been increasing with every new generation. In future, the decreasing feature size of hardware will however not lead to more reliable but to less reliable hardware—logic and memory—as [17] states. [4] 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 [4]:

- **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. That results in a greater impact of variations (in this amount of dopants) onto the electrical properties of the transistors.
- **subwavelength lithography** Nowadays, the wavelength of the light used in lithography is bigger than the produced structures—resulting in rough structures. That again 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 depending on their location on the die.

So, the uncontrollable variety of the production process 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 thus become faster unreliable. Furthermore, 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 [4].

While the number of transistors per chip is expected to increase exponentially for at least another decade, we expect that the increase in CPU cycles needed to provide the functionality of safety-critical components will be much less. Therefore, we are faced with the prospect that in future we will have plenty of CPU cycles to spare but we cannot assume that all instructions are executed correctly. The objective of SEP is to provide probabilistic guarantees that can be tuned for the demanded dependability requirements and the expected maximum failure rate of the underlying hardware. SEP will support commodity hardware while executing standard binaries without requiring their recompilation. Programs with low or no safety requirements will be executed at full speed concurrently to software encoded programs.

2 Related Work

Detection of transient and permanent hardware errors is a widely researched topic. A widespread technique to detect errors in memory are error correcting

codes (ECC) and parities. [16] and [5] demonstrate their usage to protect Itanium and Power4 processors. But soft errors can also influence the logic building blocks of a computing system. On hardware level this problem is usually tackled by replication. [2] describes lock-stepped and loosely lock-stepped processors, that is redundant processors executing the same instruction stream. Their results are compared to detect erroneous executions. These approaches are completed by redundant memory blocks, redundant communication links and redundant disks. This redundancy can also provides means to fail-over.

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.

These hardware solutions which are able to handle errors in logic building blocks are very expensive. That typically rules out the execution of non-safety-critical software on such hardware. Furthermore, they are not designed for detection of permanent design faults.

Control flow checking which can be implemented in hardware, e.g., [10] and [11], or software, e.g., [1], provides means to recognize invalid control flow for the executed program, that is execution of instructions which are not expected for the executed binary. If hardware errors do only influence processed data, these errors will not be recognized. Even if an error is detected, it is not possible to identify erroneous data as it is possible with SEP. Furthermore, most control flow checking techniques require the source or assembler code of the executed program to determine a model for the expected control flow. They do not provide adjustable guarantees.

Algorithm based fault tolerance [8, 18] and self-checking software [22, 3] use invariants contained in the executed program to check the validity of the generated results. This requires that appropriate invariants exist. These invariants have to be designed to provide a good failure detection capability.

On software level, redundant execution too is used to detect hardware errors. [13] duplicates instruction and compares their outcomes. A similar approach is taken in ED4I [14], but the duplicated instructions do not process the original data but a k -multiple of it. Thus all results of duplicate instructions have to be k -multiples of the original results. So, many hardware errors are recognizable. But faults in program loading, changed control flow and operand errors are not recognized if these occur in both program versions. Also not recognized are errors in the code comparing the two results.

Currently, much research is done on how to efficiently exploit multicore processors for fault detection [19] using them for efficient redundant execution of the same program and result comparison. Here too, design faults will influence both copies, and the comparison is critical wrt. soft errors.

The Vital Coded Processor (VCP) presented in [7] 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, that is multiplication of every data item with a constant A , 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 d to detect the usage of out-dated operands (*lost updates*).

Thus, 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 and d is the time stamp which identifies the current iteration of the executed program. Signatures are associated with variables after transforming a program into single-assignment form, i.e., every variable is assigned exactly once. Signatures for all input variables are chosen during the program development process. The signatures which are to be expected for any dependent variable including output variables can be derived using the signatures of input variables and the source code of the executed program.

In the following listing a_c , b_c , and c_c are encoded variables. The signature of the result r_c can be precomputed to be $B_a + B_b + B_c$. The code has to maintain the timestamp d . That requires correctional steps such as subtraction of the current d after an addition of two encoded variables.

```
// encoded computation of a+b+c
int f(int a_c, int b_c, int c_c, int d){
    int t_c=a_c+b_c-d;    // t_c = (A*a+Ba+d)+(A*b+Bb+d)-d
                        // t_c = A(a+b)+Ba+Bb+d
    int r_c=t_c+c_c-d;    // r_c = (A*t+Bt+d)+(A*c+Bc+d)-d
                        // r_c = A(t+c)+Bt+Bc+d
                        // r_c = A(a+b+c)+Ba+Bb+Bc+d
    return r_c;
}
```

An encoded program only processes encoded values. It has no direct access to their functional values. The encoding of input values is done by a special hardware encoder before the input is given to the main CPU. The signatures for input variables and precomputed signatures for output variables are stored in another part of special hardware. The precomputed signatures are used by the hardware-implemented checker to test the validity of generated output r_c by evaluating the following condition: $(r_c - d) \bmod A == B_a + B_b + B_c$. The functional value r_f of r_c is obtained by integer division: $(r_c - d)/A$. Code checking and decoding require that $B_r = B_a + B_b + B_c < A$ is valid. Special hardware provides the d which is used for encoding, decoding, and code correction during execution. The VCP executes programs in a loop where every run uses the same signatures but a different time stamp d .

It can easily be seen that r_c will not be a valid code word if *a*) any of the operations executed is faulty, e.g., an addition which delivers for some input values incorrect results, *b*) if an unintended operation, e.g., a subtraction instead of an addition, is executed, *c*) an operand is modified or replaced with another operand encoded using a different signature or *d*) an out-dated operand, i.e., an operand from the previous iteration which is encoded using a wrong d , is used. Errors remain undetected if they result in a multiple of A plus the correct signature for the modified variable and the current time stamp d . With high

probability, an introduced error destroys the code of any variable which depends on a variable containing an invalid code word. Further faults might rectify the code. But that is very improbable because these faults would have to create a correctly encoded word for the modified variable, i.e., a multiple of A plus the appropriate signature and timestamp d . In section 4 we compute the probability for undetected errors.

Control structures such as branches or loops are implemented in such a way that the signatures of all variables whose values depend on the control structure are defined independently of the chosen branch or the number of executed iterations. However, the signature will be incorrect if the wrong branch is taken or a wrong number of iterations are executed (see [7]).

The following pseudo code demonstrates the encoding of the unencoded if-statement `if(x>=0){y=z+x}else{y=x-y}`.

```

sigCond = sigGEZ(x_c); // if (x_f < 0) sigGEZ(<0) else sigGEZ(>=0)
if( x_c >= 0 ){
  y_c = z_c+x_c-d;      // y_c=A*(z_f+x_f)+Bz+Bx+d
} else {
  y_c = x_c-y_c+d;     // y_c=A*(x_f-y_f)+Bx-By+d
  y_c+= (Bz+Bx);      // y_c=A*(x_f-y_f)+Bx-By+Bz+Bx+d
  y_c+= -(Bx-By)      // y_c=A*(x_f-y_f)+Bz+Bx+d
  y_c+= -sigGEZ(<0)+sigGEZ(>=0); // y_c=A*(x_f-y_f)+Bz+Bx+d
                                   // -sigGEZ(<0)+sigGEZ(>=0)
}
y_c += sigCond;      // By=Bz+Bx+sigGEZ(>=0)

```

`sigGEZ` computes the signature for the comparison of $x_f \geq 0$. `sigGEZ` evaluates to two different values: one if x_f is greater-equal zero and another if x_f is less zero. The signature of y_c after executing the if-statement is defined to be $B_z + B_x + \text{sigGEZ}(\geq 0)$ —independent of the chosen branch. If any of the computations or any of the used operands is faulty, the signature of y_c will be destroyed, i.e., not be equal to $B_z + B_x + \text{sigGEZ}(\geq 0)$. The same is the case if a branch is chosen which does not match x_f 's size in relation to zero.

VCP has the following disadvantages that restrict its use: *a)* 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. *b)* The source code of all software components is needed. *c)* The signatures chosen for the input variables have to be selected in such a way that the signatures of all variables are smaller than A or corrective actions during execution have to be taken. *d)* Special hardware is required to encode input variables, to store signatures, and to check the signatures of output variables.

3 Software Encoded Processor (SEP)

The goal of 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 [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 program counter. The interpreter executes the program in an encoded fashion and thereby generates encoded outputs which are checked by another (standard) hardware unit to determine if they are valid code words (see section 3.4 on Code Checking).

In this paper we focus on the problem how to cope with unpredictable dataflow of programs. For example, for a server program that can process a diverse set of requests, it is in general not possible to know the exact dataflow beforehand because it depends on the exact sequence of requests sent to the server and also the arguments provided to these requests. Hence, in general we cannot precompute the signatures of variables. Furthermore, our interpreter approach introduces new fault types which we have to deal with: the interpreter might load wrong instruction (*loading error*) from the process image, that is the instruction does not match the current program counter (PC) or the PC itself is incorrect. Second, the interpreter might execute a wrong instruction (*selection error*) which does not match the loaded one.

We will not discuss how to encode instructions such as addition, multiplication or jumps. These instructions are assumed to be atomic with high probability, i.e., either executed completely or result in invalid code words. Furthermore, different instructions have to result in different signatures for the data items they produce. See [21] for information on encoding a CPU's instruction set.

3.1 Encoding of the Process Image and Management Data

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 only depends on the memory address and the number of executed instructions (*version*). Thus, it can be computed dynamically without knowing the executed program. Only knowledge of the number of so far executed instruction and the checked address is required.

The whole process image, that is processed data and program code, are encoded using the following code: $x_c = A * x_f + h(\text{address}, \text{version})$ where *address* is the address of the encoded memory cell and *version* the number of instructions executed since program start. A memory cell can contain a data item to be processed by the executed program or an instruction to be executed by the interpreter. This code implies that every memory cell has to be updated after each instruction to match the current instruction count. In section 3.5 we shortly introduce a more sophisticated approach which generates less overhead. Obviously, encoding will result in a higher memory consumption depending on the size of A .

Multiplication with A protects from unrecognized *data modifications* and *operation errors*. A has to be chosen in such a way that it is very unlikely that

bitflips result in valid code words. That excludes all powers and multiples of two. Forin [7] suggests the usage of prime numbers.

$h(\text{address}, \text{version})$ maps *address* and *version* to a number smaller than A . Its addition protects from *operand errors*, *operator errors* and *lost updates* as the signatures introduced by Forin [7] do. $h(\text{address}, \text{version})$ is called the signature of *address* or signature of x .

To protect from *loading* and *selection errors* we have to encode the program counter (PC) too. The PC is part of the interpreter and points to the next instruction to be executed. The following code is used: $PC_c = A * PC_f + h(PC_f, \text{version})$. The next section shows how the encoding of the PC is used to detect *loading errors*.

3.2 Execution of Encoded Programs

The following pseudocode shows the interpreter main loop which executes one instruction of the binary and changes the PC to point to the next one.

```

1 // decode PC and load instruction from memory
2 PC_f=PC_c/A;   Instruction_c=M[PC_f];
3
4 // extract execution information from instruction_c
5 OpCode = getOpcode(Instruction_c - h(PC_f, version));
6 O1 = getOperand1Addr(Instruction_c - h(PC_f, version));
7 O2 = getOperand2Addr(Instruction_c - h(PC_f, version));
8 Result = getResultAddr(Instruction_c - h(PC_f, version));
9
10 // Did a load error occur?
11 S_li = Instruction_c % A - h(PC_f, version);
12 S_pc = PC_c % A - h(PC_f, version);
13
14 // execute instruction
15 switch(OpCode){
16     case ADD:
17         *Result = *O1+*O2;
18         *Result -= h(O1, version)+h(O2, version);
19         S_op=formOp(ADD, Result, O1, O2);
20         *Result += Instruction_c/A-S_op;
21         *Result += S_li+S_pc;
22         *Result += h(Result, version+1);
23     case SUB:
24         *Result =*O1-*O2;
25         *Result -= h(O1, version)-h(O2, version);
26         S_op=formOp(SUB, Result, O1, O2);
27         *Result += Instruction_c/A-S_op;
28         *Result += S_li+S_pc;
29         *Result += h(Result, version+1)
30     case MULT:
31         ...
32 }

```

```

33 version++;
34 updateSignatures ( Instruction_c );
35 incrementPC ( ) ;

```

Line 2 loads the instruction to be executed from memory. Lines 5 to 8 extract which operation is to be executed and operand and result addresses. These addresses identify registers or memory locations. immediates are handled similarly. This information can be extracted from the encoded instruction $instruction_c$ after subtracing its signature, because then the contained information is an A-multiple of the unencoded information. Lines 11 to 12 calculate checkvalues which are later on (lines 21 and 28) added to the generated result. S_{li} checks if the loaded instruction matches the current PC, i.e., the instruction signature has to equal $h(PC_f, version)$. Thus, if no error happened S_{li} should be zero. Next, it is checked if the PC used to load the instruction was a valid codeword. In that case S_{pc} should be zero.

The switch statement uses the `OpCode` to select the code implementing the instruction matching the opcode. Its selection is checked by the lines 19/20 and 26/27. If the correct branch was chosen $Instruction_c/A-S_{op}$ should evaluate to zero. If any of the check values (S_{li} , S_{pc} and $Instruction_c/A-S_{op}$) does not, their addition will destroy the code of the result.

The remaining errors: *operand*, *operation* and *operator errors* are handled in the same way as in the VCP. Lines 17 and 24 use encoded numbers for the computations and thus are protected from *operation errors*. Lines 18/22 and 25/29 correct the signature of the result to match the signature which is expected for the result address. If an *operand*, *operator* or *lost update error* for the operands had occurred this correctional step would destroy the code of the result, since the precomputed signature used for correction would not match the actual existing one.

During computation of the result of an instruction its version is updated (see lines 22 and 29). Additionally, the versions of all other encoded values have to be updated (line 34), because otherwise they would not match the current version information `version` which is incremented in each round to protect from *lost updates* (line 33) and counts the number of executed instructions. Last, the PC is incremented by an encoded addition so that it points to the next instruction to be executed. Its encoding ensures that errors on the PC and instruction loading errors will be detected.

3.3 Program Loading

On load time, binary programs have to be transformed into the encoded process image described in section 3.1. This has to be done in a safe way without unrecognized modifications. Therefore, we expect that binaries are equipped with a cryptographic hash covering all parts which are used by the program loader to install the process image in memory. After loading and encoding the binary the loader starts a check procedure which is encoded after the principles of VCP [7]. The check procedure computes the hash of the encoded process image and

compares it to the stored hash. If they are equal, the execution can be started. Otherwise, the program has to be loaded again or an error has to be reported.

3.4 Code Checking

Encoding alone will not result in the recognition of errors. Therefore, the code has to be checked. A code word is valid if the condition $x_c \bmod A == h(\text{address}, \text{version})$ holds. *address* is the address where x_c is stored and *version* either equals the global version counter `version` or is determined using `version` and an additional data structure as described in section 3.5. In addition to checking if a memory address contains a valid code word, the validity of the PC_c has to be checked because control flow errors might only manifest in destroying PC_c 's signature.

In the VCP code checking is done by a trusted hardware part. For SEP that could be an FPGA which we expect processors will have on chip in future, the graphics processors, the data receiving party or other software encoded processors—in short anything made fault tolerant by other means. FPGAs for example can be made fault tolerant using triple modular redundancy [6]. Naturally, the code can be checked in parallel by multiple independent checkers. Each of the checkers can independently interrupt the execution of the main CPU.

When and how often code checking is performed, influences performance and the latency of error detection. Code checking has to be done at least before data becomes externally visible. This generates the smallest overhead but results in the biggest latency. On the other hand, a special variable can be used which collects all the errors which occurred. The easiest way to realize such a variable is to sum up (encoded) all results generated by executed instructions. We also have to sum up the modified PC_c to be able to detect control flow errors. The resulting value of the error collecting variable is either valid encoded if no error occurred or invalid otherwise. This variable can be checked periodically. The check will introduce an insignificant overhead, but of course after each instruction two additional encoded additions are required. But an encoded addition is one of the fastest encoded operations (see section 5).

For detecting if the interpreter has crashed or hangs, the checker has to implement a watchdog functionality. The interpreter periodically has to send an alive-sign to the checker to reset the watchdog. Therefore, the error collecting variable can be used which also ensures early detection of execution errors. Additionally, the *version* variable has to be sent to the watchdog. Thus, the watchdog can check if the interpreter does make progress.

If an invalid code word is detected or the watchdog is not reset by the interpreter, the execution can be stopped (fail-stop) or otherwise dealt with, e.g., by going back to a checkpoint with data values with correct signatures or by recomputing invalid data.

3.5 Updating the Versions

Obviously, updating the whole memory image with the new version number after each instruction would generate way too much overhead. Thus, we decided to use additional data structures to store the version of its last change for each address instead of updating the whole process image after each instruction. The used data structures have to be encoded manually. If any error happened, such as lost updates to the memory and these structures, this has to result in the return of a wrong version number for the affected memory cell. This in return will destroy the code of any memory cell updated using this memory cell.

Note, that using an unencoded data structure like a hashmap would increase the probability for lost updates: a lost update and a not updated hashmap would remain unrecognized. The version of an address has to depend on the current instruction counter **version** and the data structure. The simplest way to achieve this is a list. After each instruction, the updated address is inserted before the first list element. Subtracting the number of list items in front of an address from the current **version** results in the version information for that address. If both the update of **version** and the data structure are lost, **version** will not match PC_c . If additionally, PC_c was not updated, the affected instruction will be executed again and if that re-execution is fault-free everything is fine.

When updating an address, the list item describing the last update of that address has to be removed and the surrounding items have to be updated accordingly. Thus, each item has to store an increment which is used when computing version information. The performance of the list approach highly depends on the data locality of the executed program. In worst case, it is $O(n)$ for finding a version information and updating the list. The list can be pruned by updating the versions stored in the memory. Another less locality dependent approach are tree structures which independent of data locality provide a complexity of $O(\log(n))$.

3.6 Implementation

Our proof-of-concept implementation executes DLX-binaries. DLX is an academic RISC instruction set developed by Hennessy and Patterson [15]. For compiling, we use the DLX compiler based on gcc which is provided by UC Santa Cruz for a student project [12].

Currently, we are able to execute simple C-programs which can make use of nearly the whole DLX instruction set, apart from floating point instructions. The binaries can use bitwise logical instructions but these are not completely encoded at the moment: the actual operation is done on the functional values and not on the encoded ones. Encoding of these operations is possible but our implementation is not yet feature complete. We also provide a simple set of system calls which implement basic I/O-functionality such as `printf` and file operations.

4 Guarantees

Code words consist of n functional bits and k redundant bits with $k = \text{sizeOf}(A) + 1$. Assumed, that errors are leading to a random modification of a valid code word, such a modification will result in another valid code word and thus in an undetected error with the probability

$$p_{undetected} = \frac{\text{number of valid code words}-1}{\text{number of possible words}} \approx \frac{2^n}{2^{n+k}} = 2^{-k}.$$

To determine the probability of unrecognized errors during program execution, we have to assume a failure model for the executing infrastructure. The least restrictive assumption we can make, is that with every executed instruction an error occurs which modifies the result or used operands. Modifications can change a value to any number. The error is assumed to be uniformly distributed. This results in the following probability distribution $p(x)$ for x failures (i.e., errors which are not detected because codes are still valid), within noOfInst executed instructions:

$$p(x) = \left(1 - \frac{1}{2^k}\right)^{\text{noOfInst}-x} * \left(\frac{1}{2^k}\right)^x * \binom{\text{noOfInst}}{x}$$

The actuarial expectation of this binomial distribution gives the failure rate *FIT* (failures in 10^9 hours) for SEP under the assumed model:

$$FIT(k) = \frac{\text{noOfInst in } 10^9 \text{ hours}}{2^k}$$

The required redundancy k for a decreased failure rate increases logarithmically: To achieve a failure rate of at most 1 undetected error in 10^9 hours 72 bits redundancy are required. With a 128-bit architecture—as used in IBM’s Cell processor—32 bit numbers can be encoded using 96 bits of redundancy which can ensure a failure rate of 0 FIT even for an execution environment which injects at least one error into every result or operand.

To test the fault detection capabilities of SEP, we used our simulation-based fault injection tool FITgrind [20]. We compared the three execution variants:

encoded interpreter the used interpreter was encoded as described in section 3 using an A which added 16 bits of redundancy.

unencoded interpreter the same interpreter but no encoding (of the interpreted code, data or interpreter) was used.

native the executed program was compiled to the native architecture and executed.

FITgrind was used to inject probabilistically errors of the following types into the running interpreter or native execution: bitflips in memory and operation results and execution of different instructions to simulate address line errors. How many bits were flipped was chosen according to an exponential probability distribution. That is, mostly one bit was flipped and the probability of n -bit

flips decreased exponentially with n . For one bit flips one would expect 100% coverage since one bit flip corresponds to addition or subtraction of a power of two which cannot result in another multiple of A with the same signature.

The results of an injection run were compared with an error free (golden) run. It was checked if any erroneous output, i.e., output differing from the golden run, was generated and if an error was recognized either by the OS or the interpreter. For testing, we used a program for computing the MD5 hash of a string consisting of 10,000 characters. That we executed around 8000 times for each variant. For each run, FITgrind was initialized with another random number and thus generated different error patterns.

While the encoded interpreter produced no incorrect output at all, 4% and 9% respectively of the unencoded interpreted and native executions produced incorrect output. On the other hand, 31% of all native runs and 15% of the unencoded interpreted 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 become visible.

5 Performance

For our performance measurements we encoded 32-bit programs using a 31-bit A which doubles the memory consumption and executed the following programs:

- Computation of prime numbers up to 5,000 using the Sieve of Eratosthenes which is very computation intensive and uses a lot of multiplications whose encoding is quite expensive.
- Computing the MD5 hash of a string which has a length of 10,000. This represents a real world problem. It is a mixture of loops, branches and computations.
- A Quicksort, sorting 1,000 numbers which is not computation intensive.

The comparison of unencoded interpreted version and encoded version shows the slow down induced by encoding. The measurements were done on an AMD Athlon 64 running with a clock rate of 2200 MHz under SuSE Linux. For the prime number computation the encoded version is 25 times slower than the unencoded version. But for MD5 the slowdown is 3.4 and for Quicksort 2.2 [9].

Table 1 shows the slowdowns for some encoded 32-bit operations (31-bit A) as we measured them using the processor's time stamp counter. Additions and Subtractions are quite fast, while divisions, multiplications and comparisons induce higher overheads. That confirms slowdowns measured for our example programs: A higher slowdown is to be expected for programs which are using more multiplications, divisions and comparisons.

An encoded multiplication (using encoded 32-bit numbers) requires 128-bit arithmetic which is realized using the processor's SSE extension. Hand-crafted assembler routines proved to be much slower. Comparisons are relatively slow because the signature computation requires additional encoded subtractions and unencoded modulo operations.

operation	add	unsigned sub	div	unsigned div	mult	greater
slowdown	3	2	15	7	38	30

Table 1. Runtime comparison of encoded vs. unencoded operations.

The slowdown generated by our straight-forward interpreter implementation are very high and definitely not practicable. We expect these can be reduced to the same levels other virtual machine based approaches provide.

6 Conclusion and Future Work

SEP provides failure virtualization, i.e., it turns value failures of the underlying infrastructure into crash failures. Neither expensive, e.g., radiation hardened, hardware nor recompilation of executed programs are required. Programs for which failure virtualization is required, just have to be executed using the SEP interpreter introduced in Section 3. The interpreter itself is encoded similar to the approach of [7] and encodes the executed program at load time with a similar arithmetic code as used by Forin [7]. But in contrast to Forin’s approach, this code can be checked without the requirement to know the complete data flow of the executed program beforehand.

SEP opens up the possibility to run critical applications (in particular, fail-stop but to a more limited extend also fail-operational applications) in parallel to non-critical applications on commodity hardware which is tuned for performance instead of reliability and is in particular much cheaper than hardware designed for reliability.

SEP’s failure rate, i.e., the rate of undetected value failures, is bounded. That bound depends on the amount of redundancy used for encoding and is independent of the failure rate of the underlying hardware. As long as encoded numbers do not exceed the processor’s native word width an increase in redundancy does not increase the overhead.

Our next steps will be to switch to a more widespread architecture and to replace interpretation as far as possible with precompiled and preencoded parts to reduce the overhead generated by interpretation. Than either recompilation or preprocessing of some bytecode format, e.g., Java Bytecode, will be required. Another option would be to integrate our approach into an existing virtualization framework.

Fault injection experiments have shown that in contrast to native execution or unencoded interpretation, SEP produced no unsafe (i.e., only fail-stop) runs. But a more thorough research on the error detection capabilities of the used code is required. Especially, our knowledge about the influence of the choice of A on the Hamming distance of code words is shallow.

Another interesting question we will look at is the applicability of SEP for code safety, e.g., protection from buffer overflow attacks. As long as an attacker does not know the code parameters, it is nearly impossible for him to inject valid code.

References

1. S. Bagchi, Z. Kalbarczyk, R. Iyer, and Y. Levendel. Design and evaluation of preemptive control signature (PECOS) checking. *IEEE Trans. on Computers*, 2003.
2. David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop advanced architecture. *DSN*, 2005.
3. M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of STOC '90*, United States, 1990.
4. Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 2005.
5. Douglas C. Bossen, Joel M. Tendler, and Kevin Reick. Power4 system design for high reliability. *IEEE Micro*, 2002.
6. C. Carmichael. Triple module redundancy design techniques for virtex series FPGA. Xilinx Application Notes 197, Mar. 2001.
7. P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, Sept 1989.
8. Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Computers*, 1984.
9. Thomas Knauth. Performance improvements of the vital encoded interpreter. Großer Beleg, Technische Universität Dresden, 2006.
10. Xiaobin Li and Jean-Luc Gaudiot. A compiler-assisted on-chip assigned-signature control flow checking. In *Asia-Pacific Computer Systems Architecture Conference*, Lecture Notes in Computer Science. Springer, 2004.
11. A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors—a survey. *IEEE Trans. Comput.*, 1988.
12. Ethan L. Miller. <http://www2.ucsc.edu/courses/cmeps111-elm/dlx/install.shtml>. UC Santa Cruz, School of Engineering.
13. B. Nicolescu and Raoul Velazco. Detecting soft errors by a purely software approach: Method, tools and experimental results. In *DATE 2003*, 2003.
14. Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 2002.
15. David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
16. Nhon Quach. High availability and reliability in the Itanium processor. *IEEE Micro*, 2000.
17. L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research*, 1999.
18. V.K. Stefanidis and K.G. Margaritis. Algorithm based fault tolerance: Review and experimental study. In *International Conference of Numerical Analysis and Applied Mathematics*, 2004.
19. Cheng Wang, Ho seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of CGO 2007*, 2007.
20. Ute Wappler and Christof Fetzer. Hardware fault injection using dynamic binary instrumentation: FITgrind. In *Proceedings Supplemental Volume of EDCC-6*, October 2006.
21. Ute Wappler and Christof Fetzer. Hardware failure virtualization via software encoded processing. In *INDIN 2007*, 2007.
22. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 1997.