Software Implemented Hardware Fault Tolerance
Increasing Software Dependability with RECCO: An Evaluation

Clemens Renner

Department of Computer Science, Programming Systems (Chair V)
University of Dortmund, Germany
clemens.renner@udo.edu

Abstract

Working with commercial off-the-shelf (COTS) components especially in areas with high degrees of radiation challenges software developers in a new way. The hardware on which their programs are running is subject to single event upsets (SEUs), also referred to as bit-flips. This paper describes a source-to-source compiler which introduces redundancy to cover up for eventual memory errors. Optimization and evaluation techniques for the general method are presented and discussed.

1. Introduction

Using computer systems for performing critical tasks has become more and more routine. While on the one hand, one wants to employ every possible gain in efficiency or ease of use, on the other hand there is less willingness to pay for components that supply the methods for assuring reliability on the hardware level. COTS components are nowadays widely used in life-critical areas as for instance the control systems for aircrafts, medical services or public transport. Automated tasks in these sectors may be influenced by a somewhat hazardous environment and are therefore more likely to produce errors which in turn might prove to be dangerous to the lives of those involved.

Systems in these duties prove to be highly complex. Making the implementations stable and reliable has thus become part of the software development process. When speaking of reliability or dependability, one aspect shall be focused on in this paper: the capability of the computer system to detect and possibly correct errors arising from environmental influences and as such preventing system instabilities or losses in overall performance.

The above is commonly achieved through techniques named *Software Implemented Hardware Fault Tolerance* (SIHFT). The basic idea is to analyze the criticality of the system's parts and to introduce redundancy to those components that are more likely to be affected by errors. SIHFT, as opposed to hardware-implemented fault tolerance, requires no additional electronic components and is therefore a relatively low-cost alternative to older techniques which increase a system's reliabilty or dependability through extra hardware.

Researchers at Politecnico Torino[1] have used the engineer's approach to the described problem. They have developed a source-to-source compiler which produces a modification of the original source code exploiting SIHFT methods. The results that Prinetto et al. [2] described show a quite surprising improvement in terms of dependability.

This paper is organized as follows: section two describes the source-to-source compiler while section three will focus on the methods used for variable criticality estimation. In the fourth section, methods and examples are discussed how one can test the value of a code transformation tool like RECCO (see section 2). Section five will show some links to a related approach, whereas in section six I will try to summarize ideas and effects critically.

2. SIHFT example: RECCO

RECCO is the abbreviation for "Reliable C/C++ Compiler". It resembles a means of fully automatic source code conversion. While the interface of the tool is kept very simple, the internal analysis and code modification procedures are quite intelligent. Let us take a closer look at the workflow.

Starting off, RECCO inspects the given source code and performs a so-called *Code Reliability Analysis*. The tool examines the variables and gathers informa-

[1] http://testgroup.polito.it

tion about the structure of the source code. These two steps are more or less performed simultaneously.

When looking at the variables, the compiler estimates the criticality of each variable used, depending on how often it is referenced or changes. These dependencies are used to build up a graph that represents the gained information. A little more in-depth description of how the estimation of criticality takes place can be found in section 3.

Experience shows that source code reaches high levels of complexity even when handling fairly simple problems. In order to prepare the modifications of eventually complex source code, a program is split up into blocks. These blocks are called "branch-free" which basically means that they have one entry and one exit point, thus can be considered relatively simple. The links between these blocks are now used to build another graph which describes the program control flow. Technically, this flow graph is written to an external file employing regular expressions. During the flow analysis, each block gets an identifier, which is later used in a regular expression as for instance in *ab*c* which would mean that the program always begins with a starting block called *a* which may or may not be followed by the (repeated) execution of block *b* and finally ending with block *c*, while the execution of *a* and *c* are apparently independent of input data.

The information gathered in these preliminary steps is now used to perform the acutal code transformation. In order to make sure that the program does not deviate from the intended flow of operations, a control flow checking process is introduced and is later run concurrently to the original program. The flow is checked and validated using block signatures. Each of the branch-free blocks is assigned a computed signature which provides the means for the actual control flow checking during run-time.

The variables are handled in a step called *Variable Protection phase*. The idea is as follows: We want to look at a certain variable and need some way of finding out whether that variable is either still holding the correct value for the current moment of execution or, if the latter is not the case, we want to notice that the variable data has changed unintentionally. There are basically two levels of handling this problem, the first one being the simple duplication of the variable in question which implies keeping two synchronized copies of the data in memory at all times. If one value differs from the other, we can at least notify the user or repeat the needed calculations to get a safe value. If additional reliability is needed, it can be achieved through triplication of a variable. The principle of checking for errors stays the same but the third copy allows for automatic correction in most cases, since presumably only one of the three copies had its value altered and two copies still hold the intended value. Now, a decision is needed concerning the selection of variables to be backed up and this is where the aforementioned variable criticality graph is used. In order to make this process suitable for a variety of target applications, the user can specify the trade-off between reliability and performance. If highly reliable code is needed, the overhead introduced by multiple copies of every variable is of course greater than in the case of only moderately reliable programs where only the most critical variables are duplicated or triplicated.

During run-time, as stated before, both control flow and data integrity are checked concurrently with the actual program code to minimize the inevitable overhead in terms of needed CPU time.

2.1. A closer look at RECCO

While the formal context presented in section 3 strongly relates to *RECCO**, the description of the interface deals with the basic version of the source-to-source compiler.

As detailed in the annotated screenshot (Fig. 1), the interface is quite tidy and seems easy to use. On the left hand side of the window are the most significant controls. They decide where the focus shall be during the code transformation phase. If *Allvars* is selected in the frame named *Duplication*, all variables will be duplicated (or triplicated, if this is selected in the Settings dialog). For an optimization that allows the desired trade-off between performance loss on the one hand and increase in dependability on the other hand, the user must select *Intelligent*. Once the latter is done, the slider control below the mentioned frame can be used to set the trade-off (selecting between maximum performance and maximum protection).

The large area with white background represents a table filled with information regarding the analysed variables. Various statistics can be retrieved from that table, such as variable *Birth*, *Death*, *Incidences* and the criticality *Points* before and after rearranging and similar.

Unfortunately, I was unable to produce any usable results using the version of RECCO I had at hand. It seems that the tool will not run under Windows XP. Trying all (sensible) available compatibility options did not help either. The version information dialog

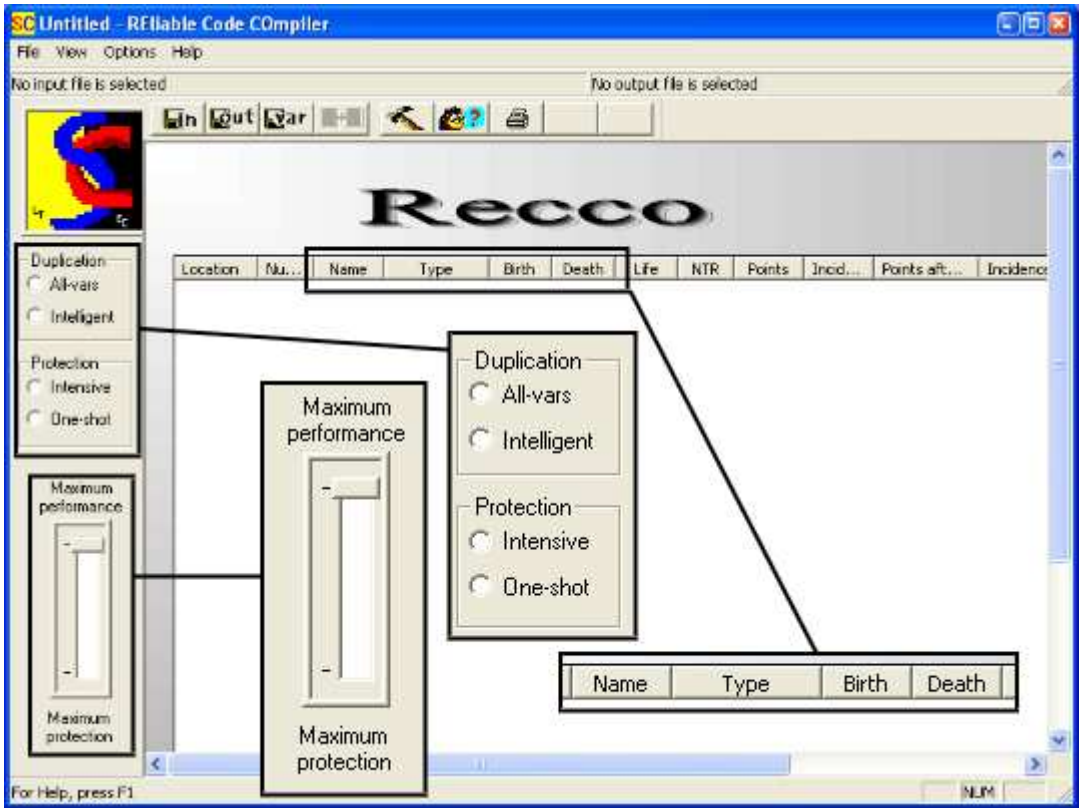

Figure 1: *The main window of the source-to-source compiler RECCO*

shows v3.0 by Luca Tagliaferri.

3. Variable Criticality Estimation

The usability of a reliability-increasing compiler is very much depending on how well the selection of so-called "critical variables" is done. Benso et al. have proposed a formal model to measure the criticality of variables [3] which shall be focused on in this section.

When speaking of variable criticality, we basically mean the probability that a variable has altered its value unintentionally. The higher the criticality, the more likely is that specific variable to change without notice. This is mostly due to the fact that certain variables have a longer lifetime than others. They get their value assigned at a certain time and offer it to anyone requesting it. For instance, if a variable gets assigned its value at the very beginning of the program execution and is only cleared when the whole program terminates and by that releases its area in the computer's memory, this exact variable is highly likely to be affected by the mentioned SEUs. This can happen during any time of program execution because the variable holds its value from beginning to end.

This is what literature refers to as variable lifetime. One attempt to resolve this issue would be to shorten the lifetime of a variable by re-ordering code in a way where the program's semantics remain unchanged. If there is a way to rearrange code segments or even individual statements so that those lifetimes can be considered optimal (i.e. lowest), it definitely reduces the probabilistic risk of being affected by SEUs. Researchers at the University of Dortmund have been looking into this problem for quite some time already and call the means and methods to re-order source code segments *Code Motion* (see [4, 5] for details or section **??** for a brief overview).

In order to avoid confusion, it should be said once more that the analysis (including eventual code manipulation based on the gathered statistics), is performed statically on the source code and there is no on-line analysis except checking for variable integrity and control flow checking.

3.1. Criticality Functions *CF* and *ICF*

But what we need in the first place is a method to determine the criticality of a variable. The definition of a *Criticality Function CF(v)* follows the one detailed

in [3]. The goal shall be to have a measure of likeliness to be affected by SEUs for the set of variables used in a program (the *CF* argument v is from that set). We will start by taking a look at which events matter when inspecting a variable. We will also see what each of these events contribute to the *Instantaneous Criticality Function ICF(v,t)* which describes the criticality of a variable v at a time t.

- *Creation (C):* Once a variable is declared in the source code, the memory allocation for that specific variable is performed which means that from this very moment on the variable might be subject to a fault in a memory segment. This not only initializes the memory for the variable but also the value of the ICF for that variable which is initially set to 0 and can be considered defined from that moment on.
- *Write (W):* This event is used to track value assignments to a variable. If we assume that the criticality of a variable does not change over time unless any *Read* operations are performed, we can define a certain ICF value of K immediately following the *Write* event remaining constant until the first *Read* event.
- *Read (R):* Whenever a variable is involved in a computation, its value is retrieved from memory. If for instance, no *Write* event occurs between two *Read* events, it is desired, of course, that the value has not changed. For the ICF, we see an increment of β after each *Read* operation since a corrupted value might be passed on to values of other variables if they are involved in a computation.
- *Last Read (LR):* As the name suggests, this event represents a variable's final Read event. From an LR event on, the variable value might still be present in memory but it can be considered insignificant since it will not be retrieved anymore. The ICF value for the variable in question will thus be set to zero since the variable content will not be used until the next *Write* or until it dies.
- *Death (D):* Represents the event of a variable being removed from memory (unallocation). A *Death* event of a variable results in the undefinition of the corresponding ICF.

We can now step up to a formal description of the Instantaneous Criticality Function where v stands for the variable in question, K represents the "basic criticality level" and β is the increment in criticality after each *Read* operation. We inspect the criticality of a given variable v at a certain instant t during the program execution. The index i shall stand for a variable (stated in [3]). We get:

$$ICF_i(v,t,K,\beta) = \begin{cases} 0 & C \leq t \leq W \\ K & W \leq t \leq R \\ \beta + ICF_{i-1}(v,K,\beta) & R \leq t \leq R \\ \beta + ICF_{i-1}(v,K,\beta) & R \leq t \leq LR \\ 0 & LR \leq t \leq W \\ 0 & LR \leq t \leq D \end{cases} \quad (1)$$

However, the given definition is inconsistent. It is not clear whether v or i actually represent the variable that is to be examined. It seems to me that the definition as shown is flawed since it also implies an ordering on the variables, as the function uses itself recursively, referring to the "previous" variable $i-1$.

I assume that the function might have been meant slightly different. As I have not found any obvious variant of the definition that is more consistent, I will explain one I found to be suitable enough. Let us think of a way to capture the point of program execution. If we are dealing with deterministic algorithms and program code, we know that for every two input instances we get the same result. Thus, we can set up a new timeline of program progress if we implement an instruction counter, for now called m. It seems that the instantaneous criticality can only be computed if we trace the program execution back to the last *Write* operation on the variable in question. This leaves us with the initial K and a certain amount of increments in steps of β. With a call of $ICF_m(v,K,\beta)$, m being the instruction counter, we can track the ICF value back to the last initial K and compute the value from there on. Thus, I would propose an alternate definition as follows:

$$ICF_m(v,K,\beta) = \begin{cases} 0 & C \leq t \leq W \\ K & W \leq t \leq R \\ \beta + ICF_{m-1}(v,K,\beta) & R \leq t \leq R \\ \beta + ICF_{m-1}(v,K,\beta) & R \leq t \leq LR \\ 0 & LR \leq t \leq W \\ 0 & LR \leq t \leq D \end{cases} \quad (2)$$

This definition seems more consistent to me.

Let us take a look at the exemplary lifetime of a variable as shown in Fig. 2. With the initial *Create* event, the ICF is defined but is set to zero since the variable does not yet hold a value. Once the first *Write* event occurs, the variable gets its value previously named K which represents the criticality until

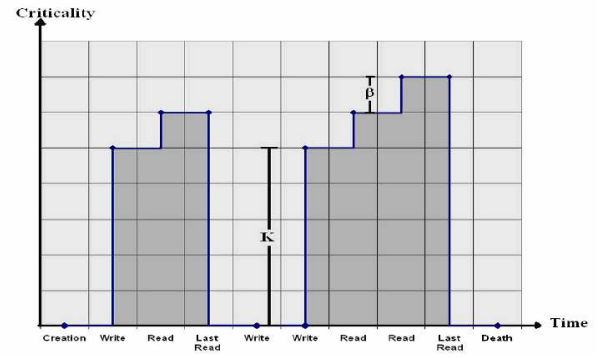

Figure 2: *ICF visualization for an exemplary variable*

a first *Read* operation. With the variable being accessed (*Read* event), its criticality increases by β to reach its "local maximum" while the *Last Read* until the next *Write* drops the criticality back to zero. Then, the variable gets its next value assigned and is back at criticality K, twice increased again by steps of β in this cycle. With the final *Last Read*, the ICF is back again at zero and gets undefined with the *Death* of the variable. As one can see in Fig. 2, the ICF is a step function.

However, it is suggested in [3] to put special emphasis on pointers, although this proposition is not explained or detailed. It is not clear from the source document whether we are considering an altered pointer target address or whether the value that is represented by the address is touched. Let us presume we are talking about an erroneous pointer address. Then, there are basically two types of targets for the pointer: In the better case, data is targeted. A slightly modified pointer address might resolve to a few steps ahead or behind of the intended register, meaning that "only" data is affected in that case. The worse case, however, is when the pointer actually targets a memory cell containing an instruction. The latter could imply a few lines of semantic nonsense. It can be assumed that these compromised instruction pointer addresses lead to erroneous results, unexpected behaviour or even crashes.

The authors of [3] introduced a pointer parameter P that should address the increased fault likeliness as stated. With a slightly different ICF, we now get a totalling Criticality Function:

$$CF(v,K,\beta,P) = \frac{1}{T_{life(v)}} \int_{T_{life(v)}} ICF(v,t,K,\beta,P)dt \quad (3)$$

3.2. Specifying the parameters

The discussed considerations are more of an analytical nature since we do not know specific values for any of K, β or P yet. In order to have a common solution for the problem of source code transformation and optimization, we need to have sensible values for the initial criticality, the criticality increment and the pointer factor.

The authors of the described ideas have taken the engineer's approach. The idea is to have a "golden application" providing for all the possible cases required for the analysis. That application is the *Fast Fourier Transfer*. This application is extended with logging mechanisms and then used as a subject to a Fault Injection Campaign (see section 4 for a description). Values of the CF (original version and fault-injected version) are compared and used for the specification of the parameters.

Starting off with an initial implementation of the mentioned application, all variable access has been logged through intercepting classes. Using a C++ template library, all access to the program's variables were redirected to the replacement C++ classes. These logged all essential events, such as creation, write, read, last read and a variable's death. Declarations of program variables have been replaced with an instance of the described "logging" class. This results in a variant of the initial program that does the same regarding the computation but is accompanied with a log file that can be used for the actual computation of the parameters.

For the logging version of the FFT application, the $CF_p(v,K,\beta,P)$ is computed. The same is then done for the version run within a Fault Injection environment and the gained results are held in a function called $CF_e(v)$ which will be described below. While the initial run represents the flawless run, unaffected by any hazardous environment, the second run simulates a program run in an environment that imposes a threat on the application's dependability.

The goal is now to compute the parameters in order to minimize the difference between the two criticality functions. In order to cover up for all possible paths taken in the program, it is vital to have a set of input data that causes the program to reach every branch at least once. Thus, 16 different input stimuli, as the authors call it, have been created for the golden application. As the CF is effectively an averaged ICF and a parametric ICF is at our disposal, we can now build a parametric CF_p:

$$CF_p(v,K,\beta,P) = \frac{1}{T_{life(v)}} \int_{T_{life(v)}} ICF_p(v,t,K,\beta,P)dt \quad (4)$$

The $CF_e(v)$ is a function gained through Fault Injection (Fault Injection will be focused on in section 4). In this "experimental Criticality Function", the criticality is expressed through the number of Fail Silent Violations (FSVs) often caused by a Single Event Upset.

Let us first collect what we have: a parametric ICF whose average is held in a parametric CF, an experimental CF with values gained through Fault Injection and presumably a difference between these two which we would like to be eliminated or at least minimized using the parameters K, β and P. Thus, a helper function containing the square deviation between the mentioned two is defined: $h(\beta,K,P) = (CF_e - CF_p)^2$. More formal:

$$h(\beta,K,P) = \sum_{i=0}^{n} \Big(CF_e(v) - CF_p(v,K,\beta,P) \Big)^2 \quad (5)$$

What needs to be taken care of now is the minimization of $h(\beta,K,P)$. This can be expressed in terms of the following constraints:

$$\begin{cases} \dfrac{\partial h(\beta,K,P)}{\partial \beta} = 0 \\ \dfrac{\partial h(\beta,K,P)}{\partial K} = 0 \\ \dfrac{\partial h(\beta,K,P)}{\partial P} = 0 \\ K \geq 0 \\ P \geq 0 \end{cases} \quad (6)$$

Solving the problem reveals specific values for the three parameters in question as follows:

$$\begin{cases} \beta^* = 4.46 \cdot 10^{-5} \\ K^* = 89.14 \\ P^* = 1.14 \end{cases} \quad (7)$$

The seemingly questionable pointer factor P finally proves a slight increment in the criticality of a variable if it is a pointer rather than a normal variable. The relatively high value for K suggests that once a variable is used, the risk to be affected by a Single Event Upset is considerably high. In other words, the time between a *Write* and the *Last Read* is most crucial to a variable's criticality. Regarding the criticality increment β, it can be said that another *Read* operation worsens the situation only slightly.

3.3. Usefulness of the approach

Two questions remain unanswered yet: It has to be shown that the values are reasonable for programs other than the "golden application" since otherwise, the expensive (in terms of time and workforce) Fault Injection process had to be repeated for each application.

Secondly, at least some empirical values need to be gathered showing an increase in dependability using the described method.

In order to answer these questions, Benso et al. [3] have considered other applications to test drive the experimental values for K, β and P. These applications are:

- *Dhrystone benchmark:* A synthetic benchmark program that measures the integer computation performance of a processor. While using the Dhrystone for benchmarking is rather uncommon nowadays, it might be useful for testing the dependability as a lot of read/write operations are performed.
- *Two custom int/float benchmarks:* The authors have developed two benchmarks of which one tends to bear variables of very different criticalities (Benchmark 1) and the other has variables with nearly the same criticality (Benchmark 2).
- *JPEG 2000:* This is an improved version of the JPEG image compression algorithm able to achieve better image quality with even higher compression ratios. Since there are many computations involved here, it is quite certain to qualify as a testing application for this case.

The procedure is similar to the one in section 3.2. The values for $CF_e(v)$ are obtained via Fault Injection (see 4.1 for details) and then held against the ones computed for $CF_p(K,\beta,P)$ using the values for the three arguments as stated before.

For all the four test applications, at least an empirical verification can be noted. In every case, the difference between the two criticalities are marginal if

existant at all. Standard deviations are 1.29 (Dhrystone), 4.82 (Benchmark 1), 6.11 (Benchmark 2) and 3.90 (JPEG 2000). Keep in mind that the initial criticality value was found to be $K = 89.14$.

The overall value of the model and the specific values can be easily seen in figure 3 where the old RECCO with the "traditional" criticality estimation is compared to the newer variant (RECCO*) which employs the presented model. The chart shows an improvement in reduction of the Fail Silent Violations (i.e. the deviations of normal program execution, resulting in program termination, but incorrect results).

However, even the new model is arguable (see section 6.

4. Evaluating the method

How can the success of the method described in the previous section be measured? In general, it is desired to have a formal proof verifying a method or a hypothesis. Since this can be increasingly difficult or even impossible, one might ask if there is another way to show at least a satisfying usability or performance.

4.1. Fault Injection

The chosen approach is Fault Injection. Let us first get a clear view on what is desired. When we are talking about dependable software solutions, we mean that the application is reliable in a way that it performs its computations correctly and completely, i.e. it does not crash. Crashes are most undesirable since they can wipe out immense amounts of work in a moment. But at least they can be noticed without effort. Feared more than a total crash is the Fail Silent Violation. While the program *seems* to work and terminate regularly, the computed values might be corrupted and may go unnoticed until further investigation in an erroneous result is done or a re-run that finishes as intended is performed. As it is all but trivial (or rather impossible, I presume) to find a formal proof for the RECCO approach, one can resort to Fault Injection.

A target application that should be tested for dependability is introduced into a Fault Injection environment. This typically means that there is a wrapper program around the application in question that provides a virtual memory layer. Thus, the program has no direct access to the machine's memory but only to the simulation. The wrapper controls what is done in the memory and is able to manipulate the registers and hereby alter the contents of the program variables. By these means, a fault can be injected into the program's memory and the effects can be observed. Usually, a Fault Injection tool provides additional logging capabilities to ease analysis. Apparently, there are three different outcomes after a fault has been injected:

- *No effect:* Although a fault has occurred (i.e. has been injected), the program follows its regular flow and produces the correct result.
- *Fail Silent Violation:* The program appears to finish regularly but produces the wrong results.
- *Crash:* The program does not finish. No usable results are produced.

Needless to say that provedly correct results are necessary to compare the Fault Injection test runs and its results with the output of the failure unaffected execution.

In order to illustrate the procedure, an example of a practical test environment shall be presented.

4.2. The Open Source Router

With Fault Injection, what can be shown aside from testing values for the dependability-increasing compiler from section 3?

One could ask about the nature of SEUs in a distributed network. Or another question might be how well the open source operating system Linux and its kernel-based routing functionalities work if exposed to a hazardous environment. As Linux finds its way into more and more business IT backbones, this should be interesting to find out about its reliability. A network is only performing as well as the routing does since today's logical net infrastructure is based on packets.

For the test environment, a simple network structure has been set up, as shown in figure 4.

Each host can be interpreted as a sub-network. Thus, the simple structure from figure 4 can be easily expanded to that shown in figure 5.

Since we are facing a distributed environment, the fault injection has to take place in a distributed fashion. This also implies that the fault injection tool itself has to be implemented on a *software* basis. The

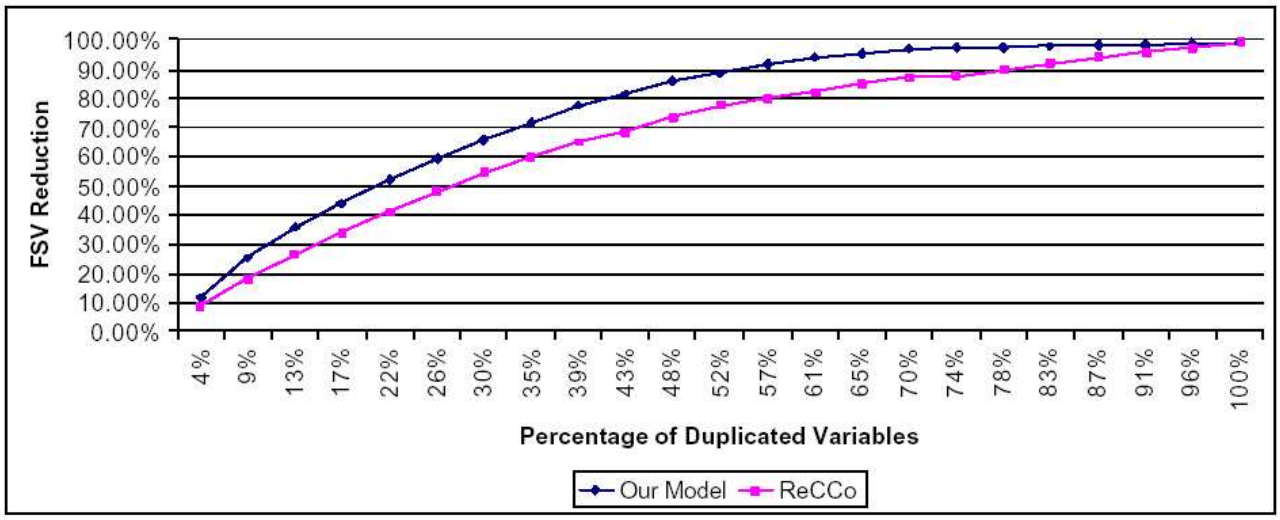

Figure 3: *Number of Fail Silent Violations, comparison between RECCO* ("Our Model") and RECCO.*

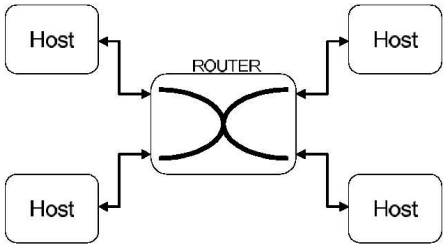

Figure 4: *Simple network structure*

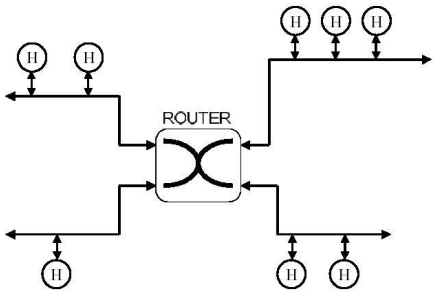

Figure 5: *Advanced network structure*

above has been achieved through a kernel daemon - a process that runs in the background, listening on specific ports and often includes some way of on-line control mechanism (e.g. a command line tool).

Figure 6 shows the structure of the UDP datagrams that are used as the test messengers. The first two fields are self-explanatory. The third field contains a host-specific counter that is incremented each time the host is sent a packet (i.e. each time the router addresses a packet to that specific host). Using this method, each station can keep an eye on how many

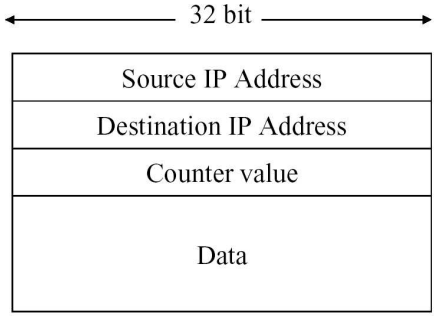

Figure 6: *Used UDP format*

packets were missing. The *Data* field basically holds a redundancy for checking the integrity of the packet. Both packet loss and integrity for the received (and missing) packets are logged.

A fault can now be injected into the router's data segment or its code segment, producing different artificial hazards.

When a fault is injected in the code segment where the instructions reside:

- *No effect:* The desired outcome.
- *Critical Fault:* Leaving the operating system alive, the router suffers a serious fault in its functionality.
- *System crash:* The routing services and all of the Linux system suffer a crash and cease to function.

When a fault is injected into the router's data seg-

ment, the overall hazard is slightly less catastrophic, at least not producing a crash. This time, the level of criticality a variable represents, influences the outcome:

- *Not critical:* Everything works as expected, all packets find their way.
- *Critical:* The router crashes eventually and the clients "report" packet loss.
- *Very critical:* The router always crashes. Routing is interrupted indefinitely.

What can be observed here is the fact that once more the code segment, when hit by an SEU, is very vulnerable and if a hazard occurs there, a system crash is close at hand. This demonstrates the need for sophisticated techniques to protect this memory segment from SEUs.

5. A related approach

As mentioned before, the longer a variable is exposed to a hazardous environment, even if it is only awaiting a read operation, the more likely it is to be affected by a Single Event Upset. So it should be quite obvious that if the variable lifetime can be shortened, the risk of an SEU is reduced.

5.1. Code Motion

Considering the way the researchers from Torino approached the problem of dependable software, what we will be talking about now might be the opposite approach. Where RECCO and the related discussion introduces redundancy to achieve a higher level of reliability, one can imagine Code Motion as the attempt to reduce the volume of vulnerability.

The basic idea of Code Motion is to eliminate partial dead code. Let us consider two almost identical graphs (Fig. 7) representing program control flow. The node labelled s shall be the starting node, whereas n designates the final node of the interesting part of the flow. The edge label $h = a + b$ shall indicate that once this edge is included in the current path from s to n, this statement is executed and involves manipulation of the variable h. It is easy to see that the three paths from the left version can be summarized into one single edge (total redundancy) whereas the right version shows one path not involving the mentioned computation.

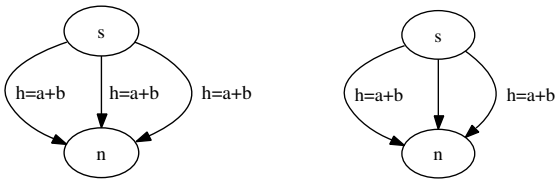

Figure 7: *Simplified program flow with total redundancy (left) and partial redundancy (right)*

The Code Motion theory now defines two predicates describing essential behaviour of the chosen Code Motion variant: $Insert_{CM}$ (basically interpreted as Insertion Points) and $Replace_{CM}$ (holding what should be inserted / replaced).

What are the major concerns? On the one hand we do not want to introduce possibly problematic code into paths on which it did not appear before. This is called *safe insertions*. Following this policy, runtime errors that did not occur before are prevented and thus a certain level of safety can be assumed. On the other hand, *correct replacements* are desired. If code is moved around the program, we still want a variable to carry its designated value since obviously otherwise semantics might change. So any newly introduced helper variables should finally still hold the intended value of the expression t in question.

Even if in abstract form, it is essential to have some type of graph representing the program control and data flow. This is achieved through program flow graphs. The node labels contain statements, so they resemble the program code. The edges in these graphs represent a possible path of execution. If a node leaves two alternative paths to follow, we find two edges leaving that node. With this graph construct, it is easy to visualize a certain case in which the Code Motion might be blocked. Whenever we find a node with more than one successor and one of these leaving edges leads to a node with more than one predecessor, we speak of *critical edges* (as depicted in Fig. 8).

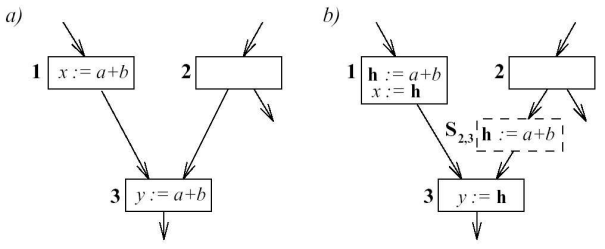

Figure 8: *Critical edges (a) and how they are bypassed (b)*

In the above figure, node 2 has two successors, one of them being node 3, which in turn has 2 predecessors (nodes 1 and 2). The partial redundancy present in the flow cannot be easily resolved since moving the computation up to node 1 would imply that it is not performed when entering node 3 on the way from node 2. The solution is to install a helper node as shown in Fig. 8b and to perform the initializations of h as shown.

There are a few more predicates we should inspect. I will not give a detailed description of these as this would imply more details on the theoretical background of the Code Motion transformations. $Comp_t(n)$ is used if the expression t is needed to compute n. Furthermore, $Mod_t(n)$ is used if t is modified upon computation of n. We will keep these at hand. So what is the purpose of Code Motion if we take a closer look?

Quality of computation: Let the quality be defined via the number of computations used per path. Then the goal shall be to minimize that number in order to achieve a higher level of quality. If we compare two Code Motion transformations, we have a formal way to designate the "better" one. Basically: *For every path starting in* s *and ending in* e*, we see fewer computations in the better Code Motion transformation.*

Lifetime optimality: As stated before, we want to avoid unnecessary Code Motions in order to not lengthen certain variable lifetimes. It is desired to shorten the lifetime of the variables in order to reduce the risk of an SEU affection - one might speak of *register pressure* in this case.

In the beginning of the research on this subject, the Code Motion technique was imperfect. The variants that were developed during the evolution of this theory shall be shortly explained:

- **Busy Code Motion:** $Insert_{BCM} = Earliest$, $Replace_{BCM} = Comp$ - While the replacement heuristic was already working as desired - only involving the needed computations - the Insertion paradigm had to be improved. If the insertion time is *Earliest* (a theoretical expression but the usual understanding fits), we create unnecessary register pressure since the variables are exposed to their environment quite long. This is not desired.
- **Almost Lazy Code Motion:** $Insert_{ALCM} = Latest$, $Replace_{ALCM} = Comp$ - Using the Busy Code Motion and delaying the insertions until the latest possible point improves the value of this variant in comparison the first approach.
- **Lazy Code Motion:** The flaw that the ALCM variant inherited was still the improper choice of insertions. While the time of insertion is optimal, the number of insertions leaves room for improvement. Instead of replacing every possible candidate, those variables that were used only after their initialization (at the point of insertion) are no longer considered. There is proof that the Lazy variant can be considered optimal [4].

One of the backbones of Code Motion ist the *Data Flow Analysis Framework*. It is used to solve underlying problems as to e.g. examine Live Variables. This is the problem of stating whether from some point on in a path of execution, a variable is used (read) again before it sees another initialization (Fig. 9) - This might be equivalent to the event of a *Last Read*, as mentioned in section 3.

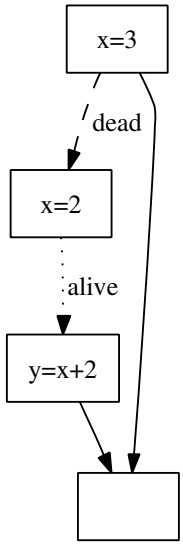

Figure 9: *Example for the Live Variables problem*

6. Summary

Before this paper concludes, I want to take the time to present some ideas I found during the making of this paper and the inspection of the source documents.

6.1. Critical thoughts

The greatest concern for me was the definition of variable criticality as found in [3]. It makes perfect sense to me that a variable is more critical than others if it is referenced more and if it is involved in more computations than others. This justifies the idea of measuring the Read operations on a variable. But isn't a variable who stays in memory for a long period of time, although rarely read, also critical in a certain way? I propose that an advanced model should take care of this aspect.

On the other hand, I think it is dangerous or at least bearing problems if a static analysis as it is done with the RECCO approach is mixed with an analysis respecting run-time effects. There might be intersections between these two ideas that complicate an automatic conversion.

The next issue are the Read and Write events itself. It appears to me that these events might change the value of a variable. Thus, one cannot be sure that the value that was read is the value that is wanted or the value that was stored in the variable at all. But this issue should be taken care of by duplication or triplication. The Write event seems more hazardous to me on the other hand. How can we be sure that the value was written correctly if we are talking about environmental hazards impacting on the normal operation of the devices.

6.2. Conclusions

In this article, I described two approaches able to increase the dependability of a given piece of software. The first one, being a source-to-source compiler, is able to perform an automatic source code conversion where variables are held several times in memory and the source code is enriched with a surveillance instance monitoring access to the variables and interfering when errors are detected and which will correct an error if possible.

The second main question was that of estimating how likely a variable is to be affected by a Single Event Upset, an error arising from the nature of the environment. A formal model was presented and discussed. For the purpose of completing the formal model and for verification of other approaches, the method of Fault Injection was described and an example about an Open Source router system was detailed.

Finally, the Code Motion theory was presented in a very brief overview, showing another approach to the problem that focuses on reducing variable lifetimes.

7. References

[1] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri: *Software Dependability Techniques validated via Fault Injection Experiments.*

[2] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri: *A C/C++ Source-to-Source Compiler for Dependable Applications*, The International Conference on Dependable Systems and Networks (FTCS-30), New York (NY), USA, June 2000, pp. 71-78.

[3] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri: *Data Criticality Estimation in Software Applications*, Proceedings, ETW 2003, IEEE European Test Workshop, Maastricht (NL), May 2003, pp. 231-236

[4] J. Knoop, O. Rüthing, B. Steffen: *Optimal Code Motion: Theory & Practice*, ACM Toplas 16(4), 1994, pp. 1117-1155

[5] J. Knoop, O. Rüthing, B. Steffen: *Partial Dead Code Elimination*, ACM SIGPLAN, 1994, pp. 147-158.

[6] http://en.wikipedia.org/wiki/Dhrystone

[7] http://en.wikipedia.org/wiki/JPEG_2000

[8] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto: *SEU Effect Analysis in a Open-Source Router via a Distributed Fault Injection Environment*, Proc. of the Conference of Design, Automation and Test in Europe, Munich, Germany, 2001, pp. 219-225.

7.1. Acknowledgements

Figure 1 is a screenshot taken from the RECCO program kindly provided by the team at Politecnico Torino.

Figures 2 and 3 were taken from [3].

Figures 4, 5 and 6 were taken from [8].

Figure 8 was taken from [4].

All images have been used to illustrate the problems in this educational article and have eventually been reproduced without explicit permission. The author of the present work does not claim any intellectual property for the images credited above.