Chapter 2

HARDENING THE DATA

1. INTRODUCTION

This chapter presents the methods for hardening a system against faults affecting the data it elaborates.

The methods exploit operation and information redundancy and are based on program modifications. The techniques described in the following paragraphs present the following general characteristics (some cases present exceptions emphasized in the specific descriptions):

- The size of the memory area containing the data is at least 2 times the size of the original program.
- The computation time of the resulting program is at least 2 times slower than the original program.
- The programmer has to follow some strict programming rules, concerning the usable data structures and statements.

This means that the adoption of these techniques is rather expensive in terms of memory size, execution slow down, and programming limitations. On the other side, they offer a very good coverage of the addressed faults.

2. COMPUTATION DUPLICATION

Computations can be duplicated at four levels of granularity: instruction, instructions block, procedure or program.

The smallest granularity is instruction-level, in which an individual instruction is duplicated. For example, the duplicated instruction is executed immediately after the original instruction is executed; the duplicated instruction may perform the same computation carried out by the original instruction, or it can even perform a mutation of the original operation.

The coarsest level of duplication is the program-level, in which the whole program is duplicated: the duplicated program may be executed after the original program completes its execution or it can be executed concurrently.

Whatever the level of granularity is adopted the technique is able to detect faults by executing a check after the duplication is executed. With the instruction-level duplication a check compares the results coming from the original instruction and its duplication; with the procedure-level duplication the results of the duplicated procedures are compared; with the program-level duplication a comparison among the outputs of the programs is executed in order to detect possible faults.

2.1 Methods based on instruction-level duplication

2.1.1 High-level instruction duplication

A simple method to achieve error detection capability is based on introducing data and code redundancy according to a set of transformations to be performed on the high-level source code [23]. The transformed code is able to detect errors affecting both data and code: the goal is achieved by duplicating each variable and adding consistency checks after every read operation. Other transformations focus on errors affecting the code, and correspond from one side to duplicating the code implementing each write operation, and from the other to adding checks for verifying the consistency of the executed operations.

The check operation is executed at every read operation in order to reduce the effect of possible error propagations.

The main advantage of the method lies in the fact that it can be automatically applied to a high-level source code [24], thus freeing the programmer from the burden of guaranteeing its correctness and effectiveness (e.g., by selecting what to duplicate and where to put the checks). The method is completely independent on the underlying hardware, and it possibly complements other already existing error detection mechanisms.

The rules mainly concern the variables defined and used by the program. The method refers to high-level code, only, and does not care whether the variables are stored in the main memory, in a cache, or in a processor register. The proposed rules may complement other Error Detection Mechanisms that can possibly exist in the system (e.g., based on parity bits or on error correction codes stored in memory). It is important to note that the detection capabilities of the rules are significantly high, since they address any error affecting the data, without any limitation on the number of modified bits or on the physical location of the bits themselves.

The basic rules can be formulated as follows:

- Rule #1: every variable x must be duplicated: let x_0 and x_1 be the names of the two copies

- Rule #2: every write operation performed on x must be performed on x_0 and x_1
- Rule #3: after each read operation on x, the two copies x_0 and x_1 must be checked for consistency, and an error detection procedure should be activated if an inconsistency is detected.

The check must be performed immediately after the read operation in order to block the fault effect propagation. Please note that variables should be checked also when they appears in any *expression* used as a condition for branches or loops, thus allowing a detection of errors that corrupt the correct execution flow of the program.

Every fault that occurs in any variable during the program execution can be detected as soon as the variable is the source operand of an instruction, i.e., when the variable is read, thus resulting in minimum error latency, which is approximately equal to the temporal distance between the fault occurrence and the first read operation. Errors affecting variables after their last usage are not detected (but do not provoke any failure, too).

Two simple examples are reported in Fig. *2-1*, which shows the code modification for an *assignment* operation and for a *sum* operation involving three variables *a*, *b* and *c*.

Original code	*Modified Code*

`a = b;`	`a0 = b0;`
	`a1 = b1;`
	`if (b0 != b1)`
	`  error ();`

`a = b + c;`	`a0 = b0 + c0;`
	`a1 = b1 + c1;`
	`if ((b0!=b1) || (c0!=c1))`
	`  error ();`

Figure 2-1. Example of code modification.

The parameters passed to a procedure, as well as the returned values, should be considered as variables. Therefore, the rules defined above can be extended as follows:

- every procedure parameter is duplicated
- each time the procedure reads a parameter, it checks the two copies for consistency
- the return value is also duplicated (in C, this means that the addresses of the two copies are passed as parameters to the called procedure).

Fig. *2-2* reports an example of application of Rules #1 to #3 to the parameters of a procedure.

Original code	*Modified code*

`res = search (a);`
`...`
`int search (int p)`
`{ int q;`
`...`
`q = p + 1;`
`...`
`return(1);`
`}`
	`search(a0, a1, &res0, &res1);`
	`...`
	`void search (int p0,int p1,int *r0,int *r1)`
	`{ int q0, q1;`
	`...`
	`q0 = p0 + 1;`
	`q1 = p1 + 1;`
	`if (p0 != p1)`
	`error ();`
	`...`
	`*r0 = 1;`
	`*r1 = 1;`
	`return;`
	`}`

Figure 2-2. Example of code transformation for errors affecting procedure parameters.

In order to assess the effectiveness of the proposed transformation rules, a set of fault injection campaigns has been reported in [25]. They have been performed on a prototypical board (called *Transputer board*) which has been originally designed for carrying out the injection of transient faults.

The Transputer board mainly includes:

- a *T225 Transputer* (a reduced instruction set microprocessor with parallel capabilities). The T225 is the main core of the board, being in charge of all the operations related with data transfer to/from the user and the implementation of test programs;
- a 4 Kbyte *PROM*, containing the executable code of the programs related with the operation of the board (boot, result transfer, program loading)
- a 32 Kbyte *SRAM*, used for the storage of T225 program workspaces, programs and data. The last 2 Kbytes are reserved to data transfer to/from the user;
- an *anti-latchup circuit*, for the detection of abnormal power consumption situations and the activation of the corresponding recovering mechanisms;
- a *watch-dog system*, refreshed every 1.5 seconds by the T225, which has been included in order to avoid system crashes due to events arising on critical targets such as the T225 internal memory cells (registers or flip-flops) or the external SRAM memory areas associated to the program modules (process workspaces).

The board can easily support fault injection experiments. Faults are randomly injected in the proper locations during the program execution. To

be consistent with the characteristics of transient errors, the injection of single faults has been performed on randomly selected bits belonging to the code and data area. The injection mechanism is implemented by a dedicated process, which runs in parallel with the tested program. The two programs (the injection program and the program under test) are loaded in the prototype board memory and launched simultaneously. The injection program waits for a random duration, then chooses a random address and a random bit in the memory area used by the program under test and inverts its value. After each injection, the behavior of the program is monitored, the fault is classified, and the results are sent to the PC acting as a host system.

The performed experiments are based on carrying out extensive fault injection sessions on three benchmark programs:

- *Matrix*: multiplication of two 10x10 matrices composed of integer values
- *BubbleSort*: an implementation of the bubble sort algorithm, run on a vector of 10 integer elements
- *QuickSort*: a recursive implementation of the quick sort algorithm, run on a vector of 10 integer elements.

For each benchmark two fault injection sessions have been executed: one on the original version of the program, the other on the modified one. Faults are injected in the memory area containing the program data. The number of faults injected in each session is 1,000 for the original and the modified versions of the program.

Faults were classified according to the categories already presented in Chapter 1.

Obviously, the goal of any fault detection mechanism is to minimize the number of faults belonging to the last category.

Table 2-1. Results of Injecting Faults in the Data Area

	Version	Effect-Less	Software detected	Failure
Matrix	Original	199	0	801
	Modified	188	812	0
Bubble Sort	Original	235	0	765
	Modified	259	741	0
Quick Sort	Original	240	0	760
	Modified	236	764	0

Table *2-1* reports the results of fault injection experiments performed on the memory area containing the data.

Note that for the original program an average percentage of 77% of faults injected in data areas led to wrong program results; on the other hand, considering the modified program, an almost equivalent average percentage

of 77% of faults are detected by the software detection mechanism and there are no faults injected in the program data that provoke failures.

Experimental results reporting average area and performance overheads for the above mentioned programs are given in [26] and are shown in Table *2-2*.

Table 2-2. Area and performance overheads with duplication and check hardening approach

Code Segment Size increase	Data Segment Size increase	Executable Code size increase	Performance Slow-down
3.64	2.0	3.4	2.92

2.1.2 Selective instruction duplication

The previous approach presents high levels of fault coverage at a cost of high memory and performance overhead. A selection of the duplicated variables and instructions can be defined in order to tune the trade-off between the level of dependability improvement and the performance degradation due to the code modification.

Reliable Code Compiler (RECCO) [27] supports the designer in identifying both the most critical portions of the code and its most critical variables, suggesting the best modifications towards a safer code. RECCO operates through the following three phases:

- *Code Reliability Analysis*: For each variable a *reliability-weight* is computed, which takes into account the variable *lifetime* and its *functional dependencies* with other variables.

  The *life period* of a variable is defined as the period starting from a write operation and ending with the last read operation on the same data preceding the next write operation or the end of the program execution. Fig. *2-3* reports a graphical representation of the life period where a_1, a_2, ..., a_n corresponds to the time instants when a given variable is accessed and *w* represents a write operation and *r* a read operation.

  The *lifetime* is defined as the sum of all the variable life periods. Data stored in variables with higher lifetime have higher probability of being corrupted, since they are stored in memory for a longer period of time. RECCO performs a static analysis of the code and evaluates the life period parameter as the number of lines of code between the write and the read operation.

  A variable *v* is *descendent* of a given variable *w* if it is written with the result of an expression which includes *w*. Variables with a lot of descendent represent a potential criticality for the system: faulty data stored in them are propagated to a large set of other variables. RECCO computes the list of descendents for each variable, analyzing the whole

program and building the correspondent *Variable Dependencies Graph* (VDG). VDG is a direct graph, in which nodes represent variables and direct edges represent variable dependencies, as shown in an example in Fig. *2-4*.

The reliability weight is computed assigning to each variable a linear function of the two parameters (lifetime and functional dependencies). RECCO sorts all the variables according to their reliability weights.

- *Code Re-ordering Phase*: RECCO modifies the original code and generate a more reliable one, functionally equivalent to the original one, but improved in terms of dependability characteristics. The adopted approach consists in performing local optimization aiming at reducing the reliability weight of the variables identified during the Code Reliability Analysis. RECCO applies the code re-ordering technique on portions of code named domains. No read/write dependencies exists among operations belonging to the same domain, i.e., inside a domain no operation reads/writes a variable that is written/read by another operation in the same domain. Therefore, within a domain all the operations can be freely re-ordered without affecting the global program behavior. Inside a given domain, each operation is labeled with a reliability weight, which is a function of the reliability weights of the involved variables. The operations are sorted for decreasing reliability weights and then rescheduled inside the domain itself, in order to minimize the whole reliability weight.
- *Variable Duplication Phase*: RECCO introduces ad-hoc modifications through the variable duplication phase, consisting in coupling some of the variables with shadow variables. The original and the shadow variable behave in the same way, storing the same type of data and being updated, with the same values, at the same time. Periodically monitoring the consistency between the two copies of the variables, it is possible to detect the occurrence of faults in one of the two replicas of the data. Variables coupled with a shadow variable are therefore reliable variables.

RECCO allows the user to trade-off between code reliability level and performance degradation, appropriately setting the reliability requirements: e.g., the user specifies the percentage of variables to be duplicated, and RECCO selects, among all the variables, the ones that are more critical for the application safety.

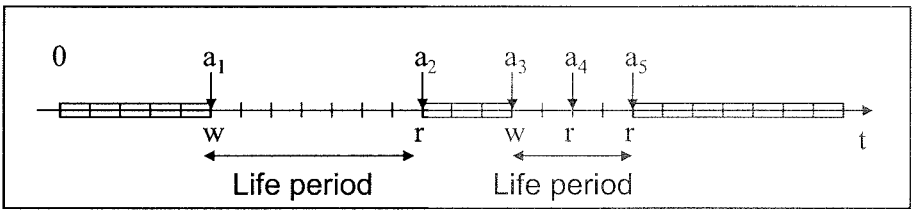

Figure 2-3. Variable's lifetime definition.

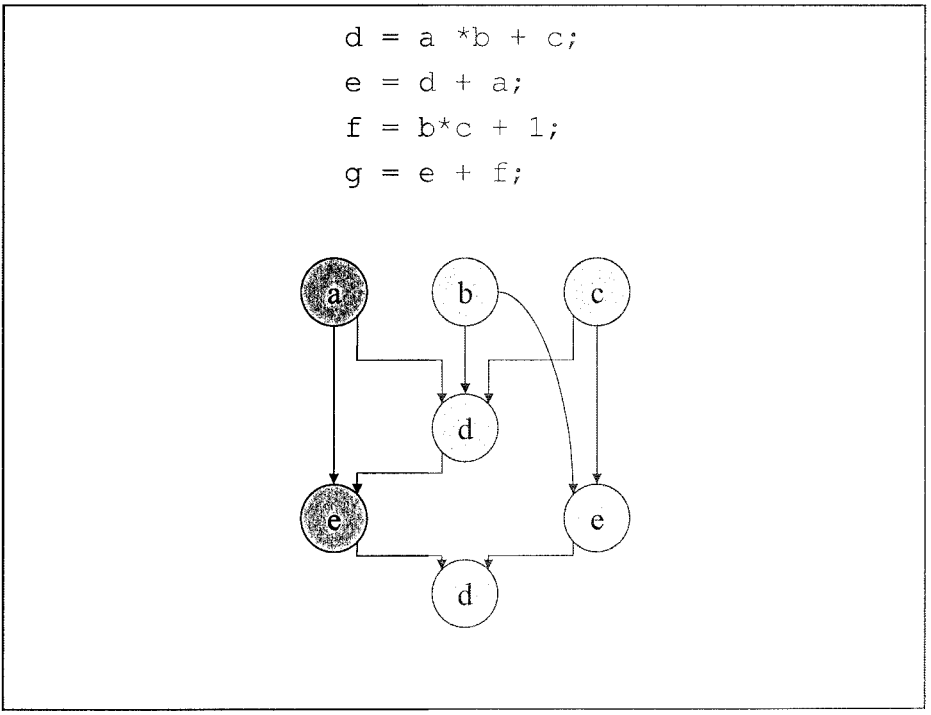

Figure 2-4. Variable Dependencies Graph.

Experimental results gathered through fault injection experiments on a set of benchmark programs demonstrate that duplicating 30% of the variables, the failures are reduced by 68% with respect to the original code; duplicating the 70% of the variables allows to reach a reduction of failures of 70%. Performance degradation and memory overhead depends strictly on the percentage of variables duplicated: performance slow down by 6% is

observed with 30% of variables duplicated; while 18% of memory overhead is needed with 30% of variables duplicated.

2.1.3 Assembly-Level Instruction Duplication

Trends in processor architecture have shown an increasing use of *Instruction-Level Parallelism* (ILP) to improve performance. In addition to pipelining individual instructions, it has become very attractive to fetch multiple instructions at the same time, and execute them in parallel to use functional units whenever possible. This form of ILP is called super-scalar execution. It provides a way to exploit available hardware resources in the system. When superscalar processors are used, it is possible to exploit ILP for error detection.

The basic idea presented by the EDDI (*Error-Detection by Duplicated Instructions*) technique [28] is to duplicate the original instructions in the original assembly source code using duplicated registers and variables, too, according to the following basic rules:

- A *master instruction* (MI) is the original instruction in the source code.
- A *shadow instruction* (SI) is the duplicated instruction added to the source code.
- General purpose registers and memory are partitioned into two groups for MI and SI instructions.
- The registers and memory for MI instructions should always have the same values as the corresponding registers and memory for SI instructions. If there has been a mismatch between a pair of registers for MI and SI, an error can be detected by comparing the values stored into the two registers. A *compare instruction* (CI) compares the values of the two registers, and invokes an error handler if they do not match.

A simple example of source code containing just one MI instruction is the following:

```
ADD R3, R1, R2              ; R3 <- R1 + R2
```

The corresponding SI and CI instructions can be the following:

```
ADD R23, R21, R22           ; SI
BNE R3, R23, gotoError      ; CI
```

Let registers R1, R2, R3 be the master registers, and R21, R22 and R23 the shadow registers that contain the same value as R1, R2, R3, respectively. The CI instruction is executed comparing the values stored in the registers containing the result of the sum (R3 and R23), and if a mismatch is found the control is transferred to an error handler (labeled `gotoError`).

The description of the method requires the following preliminary definitions. A *store* instruction is an instruction that stores the value of a variable in memory. According to this definition, a *Storeless Basic Block*

(SBB) is a sequence of instructions in which there is no store instruction except for the last one, which can be a store, or a branch instruction. An example of SBB is shown in Fig. *2-5*.

```
ADD   R1, R2, R3
SUB   R4, R1, R2
AND   R5, R1, R2
MUL   R6, R4, R5
ST    R6
```

Figure 2-5. Example of SBB.

Within a SBB, the SI instructions are scheduled to maximize resource use by attempting to use idle resources, which are not used by MI instructions. A detailed description of the scheduling algorithm is not under the scope of this book, and is presented in details in [28].

If the last instruction of an SBB is a store instruction, then a CI instruction is placed before the store instruction to compare the master and shadow values that are going to be stored in memory.

The EDDI method has been experimentally evaluated on a SGI Octane, that uses the 4-way super-scalar R10000 MIPS processor. Eight benchmark programs were used: FFT, matrix multiplication, Fibonacci, Hanoi, compress, shuffle, Quick sort and Insert sort. The method has been evaluated through a fault injection approach that forces 1 bit-flip in the code segment of the machine code. The location of the bit-flip is determined randomly for each iteration.

On average, in the original programs, 20% of the injected faults produced incorrect outputs and were not detected. On the other hand, only 1,5% of the injected faults in a program hardened with EDDI produce incorrect outputs and were not detected.

Because extra instructions are added to the original assembly code, the program with EDDI suffers from an increase in code size and loss of performance. The execution-time overhead depends on the parallelism available, too, and varies from 13% to 105%. The size overhead strictly depends on the program type, too, and varies from 44% to 113%. The authors made a comparison with [23] and showed that EDDI presents a better error detection capability, thanks to its assembly-level application. EDDI presents a finer grain error detection capability and lower latency. Consequently, it has higher chance of detecting faults that might cause cases of undetected errors that can propagate or get masked.

Detection capability can be obtained exploiting idle cycles of a fine-grain parallel architecture composed of multiple pipelined functional units, where each functional unit is capable of accepting an instruction in every clock cycle. This approach is called *instruction re-execution* [29] and addresses the performance degradation caused by time redundancy. With instruction re-execution, the program is not explicitly duplicated. Rather, when an instruction reaches the execution stage of the processor pipeline, two copies of the instruction are formed and issued to the execution units. Since instructions are duplicated within the processor itself, the processor has flexible control over the scheduling of redundant computations. Dynamic scheduling logic combined with a highly parallel execution core allows the processor to exploit idle execution cycles and execution units to perform the redundant computations. This is possible because there are not always enough independent operations in the program to fully utilize the parallel resources.

A further strategy exploiting parallelism are Very Long Instruction Word (VLIW) processors, that are becoming popular for their ability to process more than one operation per clock cycle. The intrinsic redundancy of the data path units in VLIW processor architectures provides the resources for executing the detection capability concurrently with respect to the *nominal* program (i.e., the original unhardened program).

The insertion of redundant operations for fault detection directly in the source code is not a viable approach since optimization policies tend to detect such redundancy and collapse the original and added operations into a single one, this making the modification useless. Furthermore, by acting on the compiled code, specific optimizations can be performed to minimize code growth and performance degradation.

The approach proposed by Bolchini [30] proposes a second flow of operations created and executed concurrently. This approach considers faults in the register files, thus covering faults in the processor data-path.

Duplication and comparison is the adopted redundancy scheme for achieving the desired hardware fault detection properties. Each operation concerning data path functional units is executed twice and compared in order to detect possible mismatches. The method does not provide the straightforward execution of the same operation on two different functional units, but create a similar operation on a copy of the data stored in the local memory unit. Once data are loaded from main memory, a copy is made and a parallel flow of operations is carried out on the copy, concurrently with respect to original values. A comparison of the produced values is then performed via additional software operations.

The method is based on the implementation of three key elements:

- The compiled application source code (*nominal*).

- A parallel flow of computation on a copy of the nominal values (*checking*).
- Additional operations for comparing corresponding nominal and checking results (*checker*).

The proposed approach that maintains the original hardware architecture consists in compiling the application source code on a reference architecture with one half of the hardware resources and VLIW width. This solution is transparent to both user and system. The checking and checker operations are introduced to fill the unused VLIW word and use the remaining resources. The checking code performs the same operation as the nominal one on a different subset of the register files and on different functional units. The parallel checking code is generated according to the kind of operation.

This approach provides an initial scheduling of the application code on an architecture that has one half of the actual hardware resources, but the experimental results showed the performance degradation ranges from 2% to 25% with respect to the nominal application code. The limited impact can be related to the low average number of operations per clock cycle, which leaves several empty space for duplicated operations.

Full duplication may cause an unacceptable overhead in terms of performance and energy consumption. This is particularly true for large segments of embedded markets where performance and power will continue to be as important as dependability. The approach proposed in [31] presents a technique that fills empty execution slots with duplicate instructions under a performance bound. The compiler determines the instruction schedule by balancing the permissible performance degradation with the required degree of duplication. The objective is to maximize the number of duplicated instructions with a fixed performance overhead. The algorithm considers for each instruction *i* its *duplication range* that is the range of cycles within which its duplication can be scheduled. This range is determined by the instructions that *i* depends on as well as the instructions that overwrite the register read by *i*. The duplicated instruction cannot be scheduled before the source operands for the instruction are read. The algorithm considers each instruction in turn, identifies its duplication range, and creates a duplicate for it if the duplication does not exceed the schedule length by a fixed limit. The experimental results reported figure out that the full duplication incurs an average increase of 42% in the original schedule length, while the method is able to duplicate more than 40% of the instructions without an increase in the original schedule cycles. The percentage of duplicated instructions increases as the performance bound is relaxed. As a consequence, this approach allows the designer to conduct tradeoff analyses between performance and dependability.

2.2 Procedure-level duplication

2.2.1 Selective Procedure Call

The *Selective Procedure Call Duplication* (SPCD) [32] technique is based on the duplication of the procedure execution. The major goals of this approach are the improvement of the system reliability by detecting transient errors in hardware, taking into account the reduction of the energy consumption and of the overhead.

Some industrial experimental results show that significant energy is consumed in clock circuitry and in caches. Therefore, reducing the number of clock cycles and cache access as well as memory access is important to reduce energy dissipation in the system.

SPCD minimizes energy dissipation by reducing the number of clock cycles, cache accesses, and memory accesses by selectively duplicating procedure calls instead of duplicating every instruction. The number of additional clock cycles is reduced because the number of comparisons is reduced by checking the computation results after the original and duplicated procedure execution, instead of checking the results immediately after executing every duplicated instruction. The code size is reduced because some of the procedures are not duplicated. If the code size is reduced the probability of an instruction cache miss can be lowered and energy consumption can be reduced for fetching instructions from the cache to the processor, or moving instructions from the memory to the cache. Also, reducing the number of comparisons decreases the number of data accesses to the data cache and the memory, resulting in reduced energy consumption.

However, there is a trade-off between energy saving and error detection latency: longer error detection latency reduces the number of comparisons inside the procedure and, therefore, saves energy. The shortest error detection latency can be achieved by instruction-level duplication. In procedure-level duplication, comparison of the results is postponed until after executing the called procedure twice; then, the worst case error detection latency corresponds to the execution time of the original and duplicated procedure and the comparison time.

A procedure is a sequence of statements, with an identifying name, executed as a unit through its call in any part of the program. Fig. *2-6* shows the original sample source code where procedure A calls procedure B.

```
int a,c;
void A ()
{
     a = B(b);
     c = c + a;
}
int B (int b)
{
int d;
     d = 2 * b;
     return(d);
}
```

Figure 2-6. Sample source code.

```
int a, a1, c, c1;
void A2 ()
{
     a = B2(b, b1);
     a1 = a;
     c = c + a;
     c1 = c1 + a1;
     if (c <> c1) errorHandler();
}
int B2(int b, b1)
{
int d, d1;
     d = 2 * b;
     d1 = 2 * b1;
     if (d <> d1 ) errorHandler();
     return(d);
}
```

Figure 2-7. Instruction-level duplication.

With instruction-level duplication, all the instructions in the procedures A and B are duplicated as reported in Fig. *2-7*. The code size of the procedures A2 and B2, including comparison statements, is more than twice the original code size of A and B.

A procedure-level duplication is obtained calling twice the procedure; the procedure is called with the original parameter first and then with the duplicated variable as a parameter. Fig. *2-8* shows the resulting source code: the code size of procedure A2 (containing the duplication of the called procedure B) is more than twice the original code size of A, but the size of procedure B is the same in the original and in the modified programs. As a consequence, in a procedure-level duplication program the resulting code size is lower than in an instruction-level duplication one.

```
int a, a1, c, c1;
void A2 ()
{
     a = B(b);
     a1 = B(b1);
     if (a <> a1) errorHandler();
     c = c + a;
     c1 = c1 + a1;
     if (c <> c1) errorHandler();
}
int B(int b)
{
int d;
     d = 2 * b;
     return(d);
}
```

Figure 2-8. Procedure-level duplication

If the called procedure modifies a global variable the duplicated execution of the procedure can introduce an incorrect behavior. Let consider the example shown in Fig. *2-9*, where the procedure B updates the values stored in the global variable *g*.

```
int a,c;
int g;
void A ()
{
     a = B(b);
     c = c + a;
}
int B (int b)
{
int d;
     d = 2 * b;
     g = g + 1;
     return(d);
}
```

Figure 2-9. Sample source code with a global variable modified by the called procedure.

If the procedure B is executed twice, the global variable called *g* is increased twice instead of once. In this case, as shown in Fig. *2-10*, one needs to duplicate the global variable *g1* and the duplicate procedure B1 that modifies this *g1*. The procedures B and B1 are functionally identical, except that B modifies *g* and B1 modifies *g1*.

The basic rules to be considered in a procedure-level duplication approach are the following:

- Every procedure should either be repeated twice or contain duplicated instructions. A procedure that has duplicated instructions can detect an error. A procedure that does not have duplicated instructions should be executed twice, so that an error can be detected.
- If a procedure has no duplicated instructions, all the procedures called by it should have no duplicated statements.

SPCD presents an heuristic algorithm developed to satisfy the previous rules and involving 2 objectives: reducing error detection latency and minimizing energy consumption. In particular, the algorithm presented in [32] minimizes energy consumption under a given error detection latency constraint.

SPCD was simulated with some benchmark programs.

Fault injection experiments were executed injecting single-bit flip faults in the adder unit. Experimental results show that:

- As the error detection latency increases, the energy consumption is reduced

- The data integrity (i.e., the correctness of the outputs) reported is always 100%
- The number of detected faults decreases as the error detection latency increases, but the undetected faults don't cause any failure because they don't affect the final results.

In order to evaluate the feasibility of the approach in terms of energy consumption saving, SPCD is compared with the hardened program obtained applying an instruction-level duplication approach [28]. The obtained results show that SPCD allows an energy saving of 25% with respect than the energy consumption required by an instruction-level duplication approach.

```
int a, a1, c, c1;
int g, g1;
void A2()
{
     a = B(b);
     a1 = B1(b1)
     if (a <> a1) errorHandler();
     c = c + a;
     c1 = c1 + a;
     if (c <> c1) errorHandler();
}
int B (int b)
{
int d;
     d = 2 * b;
     g = g + 1;
     return(d);
}
int B1 (int b)
{
int d;
     d = 2 * b;
     g1 = g1 + 1;
     return(d);
}
```

Figure 2-10. Sample source code with a duplicated global variable modified in the called procedure.

2.3 Program-level duplication

2.3.1 Time redundancy

Time redundancy is a technique in which a computation is performed multiple times on the same hardware. A particular application of time redundancy is the *duplication* of the processing activity as a proper technique to detect faults of the underlying hardware. A particular form of such duplication is a virtual duplex system (VDS), where the duplicity is achieved by temporal redundancy, obtained by executing two programs performing the same task with the same input data twice. Virtual duplex systems provide a cost advantage over duplex systems because of reduced hardware requirements: VDS only needs a single processor, which executes both software variants. Transient hardware errors are covered due to time redundancy, as only a single variant is affected. Permanent hardware errors are covered due to design diversity: the program variants of a VDS are diversified in order to reduce the probability that both variants are affected in the same way.

The disadvantage of time redundancy is the performance degradation caused by repetition of tasks.

There are different kinds of duplication: one option consists in running entire programs twice, whereby another option is to execute the duplicated processes in short rounds and switch between them. The switching introduces extra overhead, but can be used to compare intermediate results more frequently in order to reduce the fault latency.

The structure of a VDS is reported in Fig. *2-11*. Each version of program is called *variant*. A VDS built to calculate a specified function f consists of two diversified program variants P_a and P_b calculating the functions f_a and f_b, respectively. In absence of faults $f = f_a = f_b$ holds. If an existing fault affects only one of the two variants or both of them in different ways, then the fault can be detected comparing the results $f_a(i)$ and $f_b(i)$.

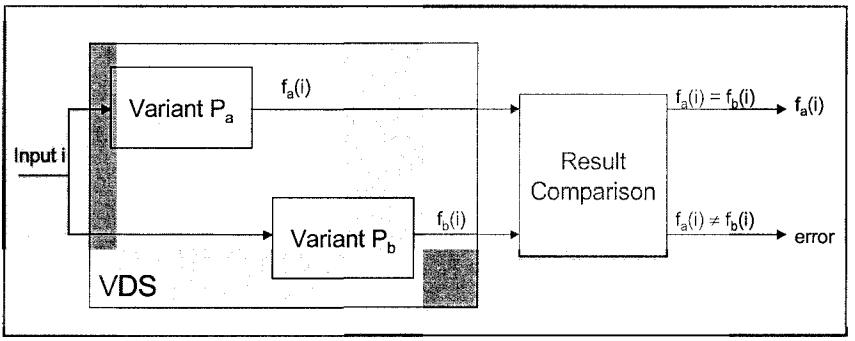

Figure *2-11. Structure of a VDS.*

The kind of faults to be detected by a VDS highly depends on the diversity techniques used to generate the VDS.

As far as VDS is considered, if, for example, two independent teams are developing different variants of a program, then the resulting VDS may have the ability to detect specification or implementation faults. If, as a second example, two different compilers are used to compile the same source code, then the resulting VDS may have the ability to detect faults stemming from compiler faults. The capability of diversified program variants to detect hardware faults has been mentioned and investigated in [33]. The basic idea is that two diversified programs often use different parts of the processor hardware in different ways with different data.

Variants can also be generated by applying manually different diversity techniques. However, some algorithmic approaches have been proposed in order to properly generate effective software variants.

In [38] a systematic method is presented based on the transformation of every instruction of a given program into a modified instruction or sequence of instructions, keeping the algorithm fixed. The transformations are based on a diverse data representation. Since a diverse data representation also requires a modification of instructions that may be executed, new sequences of instruction have to be generated, that calculate the result of the original instruction in the modified representation. The transformations are generated at the assembler and the high-level programming language. Some examples of the modification rules are:

- logical instructions can be modified according to the de Morgan Rules (e.g., `a or b = NOT (NOT (A) AND NOT (B))`)
- arithmetic instructions can be modified according to the two's complement properties (`a+b = - (-a) + (-b)`).

In [35] a method for the automated generation of variants is proposed. The tool is able to generate two different but semantically equivalent pieces

of assembler code, exploiting a set of modification rules. Some examples of modification rules are:

- Replacement of jump instructions (e.g., replacement of conditional jump instructions by appropriate combinations of other jump instructions)
- A consistent register permutation
- Substitution of a multiplication statement by a subroutine that performs multiplication in a different way.

2.3.2 Simultaneous multithreading

Simultaneous multithreading (SMT) is a novel technique to improve the performance of a superscalar microprocessor. A SMT machine allows multiple independent threads to execute simultaneously, i.e., in the same cycle, in different functional units. VDS can be effectively exploited on a SMT machine, executing two threads in parallel, shifting time redundancy to spatial redundancy [36]. Because of the improved processor utilization and the absence of a context switch the time execution is reduced with respect to the correspondent duplicated implementation on a conventional processor.

With the *Active-Stream/Redundant-Stream Simultaneous multithreading* (AR-SMT) [37] approach two explicit copies of the program run concurrently on the same processor resources as completely independent programs, each having its own state or program context. The entire pipeline of the processor is conceptually duplicated. As described in Section 2.1.3, in superscalar processors often there are phases of a single program that do not fully utilize the microprocessor architecture, so sharing the processor resources among multiple programs will increase the overall utilization. Improved utilization reduces the total time required to execute all program threads, despite possibly slowing down single thread performance. AR-SMT is based on 2 streams: active stream (A-stream) and redundant instruction stream (R-stream). The active stream corresponds to the original program thread and as instructions from the A-stream are fetched and executed, and their results committed to the program's state, the results of each instruction are also pushed on a FIFO queue called *Delay Buffer*. Results include modifications to the Program Counter by branches and any modifications to both registers and memory. The second stream (R-stream) is executed simultaneously with the A-stream. As the R-stream is fetched and executed, its committed results are compared to those stored in the Delay Buffer. A fault is detected if the comparison fails, and the committed state of the R-stream can be used as a checkpoint for recovery. Simulations made on a processor composed of 8 Processing Elements show that AR-SMT increases execution time by only 10% to 40% over a single thread thanks to the optimized utilization of the highly parallel microprocessor.

2.3.3 Data Diversity

The method exploits data diversity, by executing two different programs with the same functionality, but with different data sets and comparing their outputs. This technique is able to detect both permanent and transient faults.

This approach needs two different programs starting from the original program and transforming it into a new one in which all variables and constants are multiplied by a *diversity factor k*. Depending on the factor *k*, the original and the transformed programs may use different parts of the underlying hardware and propagate fault effects in different ways. If the two programs produce different outputs due to a fault, the fault can be detected by examining if the results of the transformed program are also *k* times greater than the results of the original program. The check between the two programs can be executed in two different ways:

1. another concurrent running program compares the results
2. the main program that spawns the original program and the transformed program checks their results after they are completed.

The program transformation changes a program *P* into a new program *P'* with diverse data in which all variables and constants are *k*-multiples of the original values when the program *P'* is executed. It consists of two transformations:

1. expression transformation
2. branching condition transformation.

The expression transformation changes the expressions in *P* to new expressions in *P'* so that the value of every variable or constant in the expression of *P'* is always the k-multiple of the corresponding value in *P*. Since the values in *P'* are different from the original values, when we compare two values in a conditional statement, the inequality relationship may need to be changed if the diversity factor is negative. For example, the conditional statement `if (i<5)` in *P* needs to be changed to `if (i > -10)` in *P'* when $k = -2$.

The branching condition transformation adjusts the inequality relationship in the conditional statement in *P'* so that the control flows in *P* and P' are identical.

The sample program in Fig. *2-12* is transformed to a diverse program shown in Fig. *2-13* where $k = -2$.

```
x = 1;
y = 5;
i = 0;
while (i < 5) {
      z = x + i * y;
      i = i + 1;
      }
i = 2 * z;
```

Figure 2-12. Sample program *P*.

```
x = -2;
y = -10;
i = 0;
while (i > -10) {
      z = x + i * y /(-2);
      i = i + (-2);
}
i = (-4) * z / (-2);
```

Figure 2-13. Transformed program *P'*.

The choice of the most suitable value for k has to satisfy two goals:

1. to guarantee data integrity, that is, to avoid that two programs produce identical erroneous outputs
2. to maximize the probability that two programs produce different outputs for the same hardware fault in order to achieve error detection.

However, the factor k should not cause an overflow in the functional units. The primary cause of the overflow problem in the transformed program is the fact that, after multiplication by k, the size of the resulting data may bee too large to fit into the data word size of the processor. For example, consider an integer value of $2^{31} - 1$ in a program (with 32-bit 2's complement integer representation). If the value of k is 2, then the resulting integer $(2^{32} - 1)$ cannot be represented using 32-bit 2's complement representation. The overflow problem can be solved by scaling: scaling up to higher precision or scaling down the original data. Scaling up the data to higher precision requires a data type with a larger size. For example, data type such as 16-bit single precision integers can be scaled up to 32-bit double precision integer data type. Scaling up may cause performance overhead because the size of the data is doubled. On the other hand, scaling

down the original data (the same effect as dividing the original data by k instead of multiplying by k) will not cause any performance overhead. However, there is a possibility that scaling down data may cause computation inaccuracy during the execution of the program. In this case, when the scaled down values are compared with the original values, only the higher order bits, that are not affected by scaling down, have to be compared.

A first method [38] proposed to consider k = -1, i.e., data are complemented.

The method proposed in [39], called ED^{4}I (*Error Detection by Diverse Data and Duplicated Instructions*), demonstrated that, in different functional units, different values of k maximize the fault detection probability and data integrity (for example the bus has the highest fault detection probability when k = -1, but the array multiplier has the highest fault detection probability when k = -4). Therefore, programs that use a particular functional unit extensively need preferably a certain *diversity factor k*. Considering six benchmark programs (Hanoi, Shuffle, Fibonacci, Lzw compression, Quick sort, Insert sort), the most frequently used functional units are adders and k = -2 is the optimum value. On the other hand, the matrix multiplication program extensively uses the multiplier and the optimum value is k = -4.

The hardening technique introduces an memory overhead higher than 2 times the memory required for the original program and the performance overhead is higher than 2, too.

ED^{4}I is applicable only to programs containing assignments, arithmetic operations, procedure calls and control flow structures, and cannot applied to statements executing logic operations (e.g., Boolean functions, shift or rotate operations) or exponential or logarithmic functions.

3. EXECUTABLE ASSERTIONS

The method is based on the execution of additional statements that check the validity of the data correspondent to the program variables.

The effectiveness of executable assertions is highly application dependent. In order to develop executable assertions, the developers require extensive knowledge of the system.

Error detection in the form of executable assertions can potentially detect any error in internal data caused by software faults or hardware faults. When input data arrive at a functional block, they are subject to executable assertions determining whether they are acceptable. Output data from computations may also be tested to see if the results seem acceptable.

The approach proposed in [40] describes a rigorous way of classifying the data to be tested. The two main categories in the classification scheme are *continuous* and *discrete* signals. These categories have subcategories that further classify the signal (e.g., the continuous signals can be divided into monotonic and random signals). For every signal class a specific set of constraints is set up, such as boundary values (maximum and minimum values) and rate limitations (minimum and maximum increase or decrease rate), which are then used in the executable assertions. Error detection is performed as a test of the constraints. A violation of a constraint is interpreted as the detection of an error.

Executable Assertion and best effort recovery are proposed in [41], considering a control application. The state variables and outputs are protected by executable assertions to detect errors using the physical constraints of the controlled object. The following erroneous cases can be detected:

- if an incorrect state of the input variable is detected by an executable assertion during one iteration of the control algorithm, a recovery is made by using the state backed-up, during the previous iteration of the computation. This is not a true recovery (as we will see in Chapter 4), since the input variable may differ from the value used in the previous iteration. This may result in the output being slightly different from the fault-free output, thus creating a minor value failure (*best effort recovery*).
- If an incorrect output is detected by an executable assertion, recovery is made by delivering the output produced in the previous iteration. The state variable is also set to the state of the previous iteration that corresponds to the delivered output. This is a best effort recovery, too, since the output could be slightly different from the fault-free value.

Executable assertions with best effort recovery has been experimentally applied on a embedded engine controller [41]. Fault injection experiments executed on the original program showed that 10.7% of the bit-flips injected into data cache and internal register of a CPU caused a failure in the system. Fault injection experiments run on the hardened program modified with the executable assertions with best effort recovery showed that the percentage of failures is decreased to 3.2%, demonstrating that software assertions with best effort recovery can be effective in reducing the number of critical failures for control algorithms.

4. REFERENCES

23. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "Soft-error Detection through Software Fault-Tolerance techniques", Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1999, pp. 210-218
24. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "A source-to-source compiler for generating dependable software", IEEE International Workshop on Source Code Analysis and Manipulation, 2001, pp. 33-42.
25. P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors", IEEE Transactions on Nuclear Science, Vol. 47, No. 6, December 2000, pp. 2231-2236
26. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "An experimental evaluation of the effectiveness of automatic rule-based transformations for safety-critical applications", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2000, pp. 257-265
27. A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications", IEEE International Conference on Dependable Systems and Networks (DSN), 2000, pp. 71-78.
28. N. Oh, P.P. Shirvani, E.J. McCluskey, "Error Detection by Duplicated Instructions In Super-scalar Processors", IEEE Transactions on Reliability, Vol. 51, No. 1, March 2002, pp. 63-75
29. G. Sohi, M. Franklin, K. Saluja, "A study of time-redundant fault tolerance techniques for high-performance pipelined computers", 19-th International Fault Tolerant Computing Symposium, 1989, pp. 463-443
30. Bolchini, C., "A software methodology for detecting hardware faults in VLIW data paths", IEEE Transactions on Reliability, Vol. 52, No. 4, Dec. 2003, pp. 458-468
31. J.-S. Lu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, "Compiler-directed instruction duplication for soft error detection", Proceedings of Design, Automation and Test in Europe, 2005, pp. 1056-1057
32. N. Oh, E. J. McCluskey, "Error Detection by Selective Procedure Call Duplication for Low Energy Consumption", IEEE Transactions on Reliability, Vol. 51, No. 4, December 2002, pp. 392-402
33. K. Echtle, B. Hinz, T. Nikolov, "On Hardware Fault Detection by Diverse Software, Proceedings of the 13-th International Conference on Fault-Tolerant Systems and Diagnostics," 1990, pp. 362-367
34. H. Engel, "Data flow transformations to detect results which are corrupted by hardware faults", Proceedings of IEEE High-Assurance Systems Engineering Workshop, 1996, pp. 279-285
35. M. Jochim, "Detecting processor hardware faults by means of automatically generated virtual duplex systems", Proceedings of the International Conference on Dependable Systems and Networks, 2002, pp. 399 – 408
36. S. K. Reinhardt, S.S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," Proceedings of the 27th International Symposium on Computer Architecture, 2000, pp. 25-36
37. E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors", 29-th International Symposium on Fault-Tolerant Computing, 1999, pp. 84-91

38. H. Engel, "Data Flow Transformations to Detect Results which are corrupted by hardware faults", Proceedings of the IEEE High-Assurance System Engineering Workshop, 1997, pp. 279-285
39. N. Oh, S. Mitra, E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, Vol. 51, No. 2, February 2002, pp. 180-199
40. M. Hiller, "Executable assertions for detecting data errors in embedded control systems", Proceedings International Conference on Dependable Systems and Networks, 2000, pp. 24-33
41. J. Vinter, J. Aidemark, P. Folkesson, J. Karlsson, "Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery", Proceedings of the International Conference on Dependable Systems and Networks, 2001, pp. 347-356

http://www.springer.com/978-0-387-26060-0

Software-Implemented Hardware Fault Tolerance

Goloubeva, O.; Rebaudengo, M.; Sonza Reorda, M.; Violante, M.

2006, XIV, 228 p., Hardcover
ISBN: 978-0-387-26060-0