Compiler-level Implementation of Single Event Upset Errors Mitigation Algorithms

Adam Piotrowski, Szymon Tarnowski
Department of Microelectronic and Computer Science
Technical University of Łódź
ul. Wólczańska 221/223
90-924 Łódź, Poland
email: komam@dmcs.pl

***Abstract*—Single Event Upset is a common source of failure in microprocessor-based systems working in environment with increased radiation level especially in places like accelerators and synchrotrons, where sophisticated digital devices operate closely to the radiation source. One of the possible solutions to increase the radiation immunity of the microprocessor systems is a strict programming approach known as the Software Implemented Hardware Fault Tolerance. Unfortunately, a manual implementation of SIHFT algorithms is difficult and can introduce additional problems with program functionality caused by human errors. In this paper author presents new approach to this problem, that is based on the modifications of the source code of the C language compiler. Protection methods are applied automatically during source code processing at intermediate representation of the compiled program.**

***Index Terms*—Software implemented hardware fault tolerance, Compilation techniques, Table protection algorithm, Single event upset, Radiation tolerant system, Radiation environment**

I. Introduction

Soft error or hardware transient fault appears in electronic device when highly energized particle strikes sensitive region of circuit. This phenomena can cause no observable effects, transient disruption of the system operations or a change of logic state in data stored in the memory cells. Soft errors do not permanently damage the hardware, but cause unpredictable behavior of computer-based systems and consequently lead to loss of functionality. They are a source of problems in electronics working not only in a radioactive environment like accelerators or cosmic space, but also at a terrestrial altitude [1], [2]. According to the National Aeronautics and Space Administration (NASA) "radiation induced errors in microelectronic circuits caused when charged particles lose energy by ionizing the medium through which they pass, leaving behind a wake of electron-hole pairs [3]" are known as SEU's - Single Event Upsets. A change of single bit, induced by the radiation is called SBU - Single Bit Upset. On the other hand, if more than one bit was affected a Multi Bit Upset (MBU) has occurred. Effects when several SEUs lead to disruption of system functionality are called SEFIs - Single Event Functional Interrupts. Besides artificial radiation environments like accelerators or nuclear reactors three natural sources can cause soft errors: alpha particles from natural radioactive impurities in the device materials, high energy cosmic rays, and secondary radiation induced from the interaction of cosmic rays and boron [4], [5].

Semiconductor devices are more and more sensitive to the radiation because of increasing demand for higher density and lower voltage. Therefore, the problem of radiation influence has to be taken into consideration during each stage of the system development. Several techniques to protect devices from soft errors have been designed. Four main group of radiation mitigation techniques can be distinguished:

- hardening on the design stage,
- shielding,
- techniques implementing radiation-tolerant solutions at the circuit or system level,
- strict programming approach to fault tolerance called Software Implemented Hardware Fault Tolerance.

Detail analyze of three first methods is beyond the scope of this paper therefore next subsections address only software-based radiation protection algorithms.

II. Software-based Radiation Protection

Software-based radiation protection techniques are introduced to the original source code of application – manually during the development of software [6], [7] or automatically during the compilation of program [8], [9]. They enable a system to tolerant software faults induced by the interaction between radiation and hardware components of the system. When the faults occurs, they provide a mechanism to the software to prevent system failure from occurring. Software-based radiation protection provides services by typically using variable duplication, control sums and redundancy at instruction, source code blocks, procedure or entire program level. This approach can be used in nuclear powers, aerospace, health care or telecommunication.

According to this description, program not only has to satisfy functional specification, but also has to use special algorithms to monitor functionality, detect, signal and correct hardware errors. It is a strictly software approach, it can be implemented either in a source code written in high level programming language or assembler. It could be used with unhardened, commercial of-the-shelf components [10].

Two types of protection algorithms can be distinguished:

- data protection algorithms,

- control flow checking algorithms [11]

Algorithms belonging to the first category are described in next parts of this paper, while the second type algorithms at this moment are beyond the scope of our interest.

III. Hardening The Data

A. State of The Art

Several software methods for hardening a system against faults affecting the data were developed. In most cases, they exploit instruction and information redundancy and are based on program modifications. Redundancy can be introduced at four levels of granularity: instruction, instruction block, procedure and program. At the instruction level individual operation called master instruction is duplicated and so called shadow instruction is introduced. Both sets of source code are executed and results are compared. In the case of inconsistency, appropriate error recovery function must be executed. In second level of granularity, selected parts of program i.e. basic blocks determined in program source code are duplicated. With the procedure level duplication, results of duplicated procedures are compared. In last approach, outputs of two copies of an entire program are compared in order to detect possible faults. Original program and its copy can be executed concurrently or one after another depending on available hardware resources. In this paper only duplication on instruction-level will be taken into consideration.

First of presented protection algorithm belongs to the High Level Instruction Duplication (HLID) group [12]. This method is based on data and instruction redundancy and covers the set of source code transformation that follows three fundamental rules:

- every variable in program must be duplicated,
- every write operation has to be performed on both, copies of the variable,
- after each read operation on variable, checking for consistency has to be done. In the case of inconsistency an appropriate error recovery procedure has to be executed.

This basic rules have to be applied not only to statements like assignments or arithmetical operations but also for every procedure call in order to protect passed parameters as well as returned value. Method can be used with high level source code, but it does have the disadvantage of introducing large number of additional code, in particular conditions and brunches.

Second solution to protect microprocessor-based system against transient error is called Error-Detection by Duplicated Instructions (EDDI)[13] and belongs to the Assembler-Level Instruction Duplication group. Every instruction in assembler source code is duplicated, different set of registers must be used in both operations. In the case of store or conditional branch instruction, appropriate registers are compared and in the case of data inconsistency error handler procedure is invoked. The store is an instruction that store the value of variable in memory. Additionally, to increase program efficiency, several instruction scheduling algorithms can be used. The main disadvantage, apart from increase in the code size and lost of performance, is necessity to the assembler level implementation. For that reason, method is target-dependent and must be separately adopted for different processor families.

B. Theoretical Background

The following subsection presents a number of definitions essential for understanding proposed new data hardening algorithm.

Definition 1: A basic block (node in control flow graph) is a maximal sequence of consecutive instructions with the properties, that flow of control can only enter the basic block through the first instruction and will leave the block without halting or branching. *A recovery basic block* is a modified version of standard basic block, where function call statement is boundaries for block and jump instructions together with function call constitute separate type of block called the *jump block*.

Definition 2: Control flow graph is a directed graph where nodes correspond to the basic blocks and edges correspond to control transfers between basic blocks. In the flow graph, two nodes have special properties. Entry is a point where procedure starts, thus, it is node with no predecessor. Exit is a point where procedure exits, therefore, it is a node with no successors [14]. *Recovery control flow graph* is a control flow graph where nodes are recovery basic blocks and jump blocks. Example of recovery control flow graph is presented in Fig. 1.

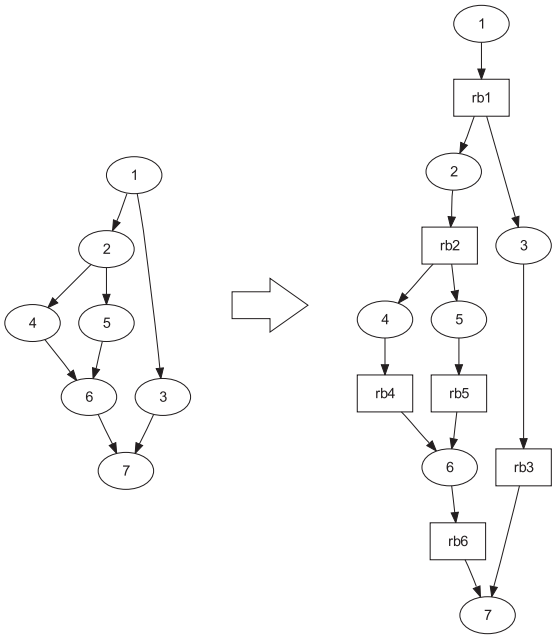

Fig. 1. Example of control flow graph and recovery control flow graph

Definition 3: A variable or a temporary is said to be *defined* when it is assigned a value, that is, when the variable or temporary is a destination of instruction. A variable is said to by *used* when it appears as a source operand in instruction. The *last use* of a variable is a program point or instruction

where the variable is used for the last time in the program or used for the last time before is it redefined. The *live range* of a variable starts from its definition and ends at its last use. A variable is said to be *live* during its live range [15]. An example of live range is presented in Fig. 2.

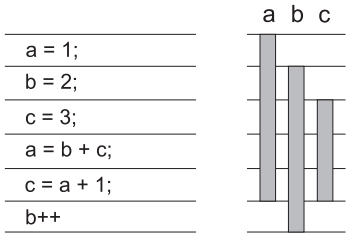

Fig. 2. An example sequence of source code and appropriate variables live ranges.

Definition 4: Lets take into consideration two instructions I_1 oraz I_2. The instruction I_2 is dependent on I_1 if and only if there is a path in the flow graph from I_1 to I_2 and the instructions might reference the same memory location [16]. Four types of dependence can be distinguished:

- The dependence is *true dependence* if I_1 is a store operation and I_2 is a load operation,
- The dependence is an *antidependence* if I_1 is a load operation and I_2 is a store operation,
- The dependence is an *output dependence* if both instructions are store operations,
- The dependence is an *input dependence* if both instructions are load instructions

IV. Recovery Instruction Duplication

Recovery instruction duplication (RID) algorithm presented in this paper is combination of both earlier introduced approaches. On one hand, program transformations are implemented in the high level source code, similarly to the High Level Instruction Duplication method. On the other hand, data consistency checking is performed only if additional conditions are fulfilled, like in the Error-Detection by Duplicated Instructions algorithm.

RID method is based on backward recovery approach [17]. It attempts to return the system to a error-free state by rolling back or restoring the system to previously saved correct conditions. System states are recorded at the recovery checkpoints selected during the source code compilation. In the case of error detection, the system state is restored to last saved point and program execution is continued from the checkpoint. Algorithm is implemented at the recovery basic blocks level, therefore checkpoints are introduced to the source code at the beginning of each block and data consistency checking is performed at the end of each block, see Fig. 3. Original instructions are duplicated and so called shadow instructions are introduced to program. Results of computation of both copies of variables that are live at the end of recovery basic block, are compared and in the case of data inconsistency, previous state of each of variable is

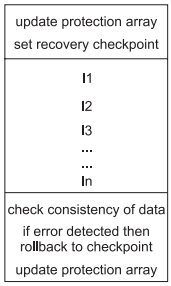

Fig. 3. Location of checkpoint and data consistency checking procedures in recovery basic block

restored and recovery mechanism is executed. Copy of local variables required to perform rollback procedure are stored in the one-dimensional array called the *recovery array*. To increase reliability of algorithm, storage area is additionally guarded by Array Protection Algorithm, described in details in [8] and [9]. At the end of block, consistency of backup copies is checked and contents of array is updated. Dead variables are removed, new live variables are inserted. The size of recovery array is equal to the maximum number of live variables existing in parallel at the beginning of the recovery blocks with properties that next use is inside block. Therefore advanced data flow analyze is required. For example array size for source code presented in Fig. 2 is zero and in Fig. 4 is one. Example implementation of RID algorithm is presented in

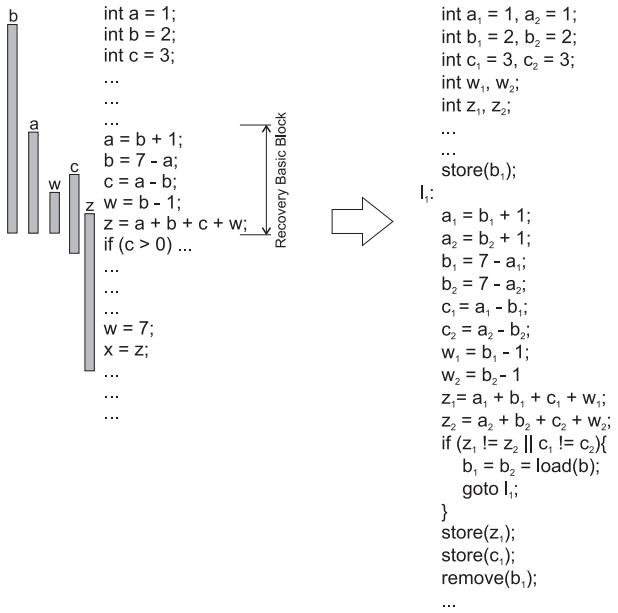

Fig. 4. An example implementation of Recovery Instruction Duplication algorithm. On the left side – input source code with marked variables live ranges and recovery basic block boundaries, on the right side – source code after algorithm implementation.

Fig. 4. Variable *a*, *b* and *c* are live at the beginning of recovery basic block. However, only *b* has to be stored in recovery array because first use of variable *a* and *c* is after new definitions in block. Based on dependency analyze one can deduce that the

rest of instruction depends on the definition of variable *b* that is located outside of currently processed block. Additionally, informations about dependencies between instructions can be used to select minimal group of variable stored into recovery array. Functions: *store*, *load* and *remove* are responsible for adding, reading and deleting specified variable from storage area with the use of Array Protection Algorithm approach. Variable *b* is dead outside of the block, therefore it must be removed from recovery array. On the other hand, *z* and *c* are new live variables and consequently they are inserted into storage area. Every instructions are duplicated and at the end of block, consistency of live variable – *z* and *c* – is checked. In the case of difference between the copies, original values of input variables are restored – in analyzed example there is only one variable *b* – and instructions in block are executed ones again. Variables live between several blocks are stored and protected into recovery array.

V. Conclusions

Recovery Instruction Duplication algorithm, presented in this paper, represents alternative approach to the problem of local variable protection. The main difference between Recovery Instruction Duplication and High Level Instruction Duplication algorithm is fact, that in first approach value consistency checking is performed only for limited set of variables. This feature is very important if algorithm is implemented at intermediate representation of source code used by compiler. Representation like three-address code allows to use in one statement assignment and one additional operator with maximum two arguments. Therefore, several temporary variables for intermediate computation results must be introduced. In most cases, temporaries are available only in one block of source code. Consistency checking performed for every temporal variable will significantly increase the size of program source code. In RID algorithm temporaries are protected only if they are used outside of one recovery basic block. In both approaches every instructions are duplicated and operations are performed on two sets of variables.

Algorithm RID has been adopted to implementation in source code written in high level programming language. For that reason and in contrast to Error-Detection by Duplicated Instructions, it is independent on the targeted hardware architecture. In both approaches consistency checking is postponed, in EDDI algorithms it is performed only for operations that write data to memory, in RID it is done only for live variable.

The main drawbacks of presented method are increase of a final code size and a decrease of program efficiency. Both disadvantages result from additional operations like comparisons and instruction duplication inserted into program to increase reliability of the system. Nevertheless, this is characteristic feature of every algorithms based on the redundancy. Advanced data flow and control flow analyze can be the source of information that allows to decrease this negative influence on protected program.

Presented algorithm can be treated as a form of temporal redundancy. In the case of error, the same source code is executed ones again. Simple correction of data and reuse of the same software can overcome transient faults induced by the radiation. On the other hand, temporal redundancy introduce unpredictable delays to the application. Therefore applications with hard real-time constrains are not good candidates for this solution.

Acknowledgment

The research leading to these results has received funding from the European Commission under the EuCARD FP7 Research Infrastructures grant agreement no. 227579 and Polish National Science Council Grant 642/N-TESLA-XFEL/09/2010/0.

The authors are a scholarship holders of project entitled ”Innovative education ...” supported by European Social Fund.

References

[1] J.F. Ziegler, “Terrestrial cosmic ray intensities,” *IBM J. Res. Dev.*, vol. 42, no. 1, 1998.

[2] S.E. Michalak and K.W. Harris and N.W. Hengartner and B.E. Takala and S.A. Wender,, “Predicting the number of fatal soft errors in los alamos national laboratory’s ASC Q supercomputer,” *IEEE Transactions on Device and Materials Reliability*, 2005.

[3] National Aeronautics and Space Administration, *NASA Thesaurus vol.1*, 2007. [Online]. Available: http://www.sti.nasa.gov/thesfrm1.htm

[4] R. C. Baumann, “Soft errors in advanced semiconductor devices — part I: the three radiation sources,” *Device and Materials Reliability, IEEE Transactions on Volume 1, Issue 1*, 2001.

[5] ——, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Transactions on Device and Materials Reliability, Vol. 5, No. 3,*, 2005.

[6] M. Rebaudengo and M. Sonza Reorda and M. Violante, “A new approach to software-implemented fault tolerance,” *IEEE Latin American Test Workshop*, vol. 40, pp. 433–437, 2002.

[7] M. Rebaudengo and M. Sonza Reorda, “Evaluating cost and effectiveness of software redundancy techniques for hardware errors detection,” *FTCS-28, The 28th Annual International Symposium on Fault-Tolerant Computing, June 23-25, Munich (Germany)*, pp. 88–89, 1998.

[8] A. Piotrowski and D. Makowski and G. Jabłoński and S. Tarnowski and A. Napieralski, “Hardware fault tolerance implemented in software at the compiler level with special emphasis on array-variable protection,” *MIXDES 2008 - Mixed Design of Integrated Circuits and Systems, 19-21 June, Poznan (Poland)*, 2008.

[9] A. Piotrowski and D. Makowski and G. Jabłoński and A. Napieralski, “The automatic implementation of software implemented hardware fault tolerance algorithms as a radiation-induced soft errors mitigation technique,” *Nuclear Science Symposium, Medical Imaging Conference and 16th Room Temperature Semiconductor Detector Workshop23-27 June, Dresden (Germany)*, 2008.

[10] A. Piotrowski and D. Makowski and Sz. Tarnowski and A. Napieralski, “Radtest - Testing board for the software implemented hardware fault tolerance research,” *MIXDES 2007 - Mixed Design of Integrated Circuits and Systems, June 21-23, Ciechocinek (Poland)*, 2007.

[11] O. Goloubeva and M. Rebaudengo and M. Sonza Reorda and M. Violante, “Soft-error detection using control flow assertions,” *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT’03)*, p. 581, 2003.

[12] ——, *Software-Implemented Hardware Fault Tolerance*. Springer Science+Business Media, LLC, 2006.

[13] N. Oh, “Software implemented fault tolerance,” Ph.D. dissertation, Stanford University, 2000.

[14] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[15] Y. N. Srikant and P. Shankar, *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC, 2001.

[16] R. Morgan, *Building an Optimizing Compiler*. Digital Press, 1997.

[17] L. L. Pullum, *Software fault tolerance techniques and implementation*. Artech House, Inc., 2001.