Automatic Installation of Software-based Fault Tolerance Algorithms in Programs Generated by GCC Compiler

Adam Piotrowski

***Abstract*—The problem of designing radiation-tolerant devices working in application critical systems becomes very important especially if human life depends on the reliability of control mechanisms. One of the possible solution of this problem are pure software protection methods. They constitute different category of techniques to detect transient faults and correct corresponding errors. Software fault tolerance schemes are cheaper to implement since they can be used with standard, commercial of-the-shelf (COTS) components. Additionally, they do not require any hardware modification. In this paper, author propose a new implementation mechanism for software based fault protection algorithms performed automatically during application compilation.**

***Index Terms*—Single Event Upsets, GCC Compiler, software based fault tolerance**

I. Introduction

SINGLE Event Effects are a serious problem for the electronic components working not only in a hostile environment like accelerators or cosmic spaces but also are a common source of hardware failures in modern desktop computers, servers or computing clusters. Semiconductor devices are more and more sensitive to the radiation because of increasing demand for higher density and lower voltage. Particles passing through matter lose energy through a several type of interactions. Two major consequences of energy transfer from radiation to electronic materials are: atomic displacement and ionization [1]. The former occurs, when particle removes an atom from its regular position, while the latter when radiation creates an electron-hole pairs. Both phenomena lead to serious degradation effects in the device parameters, disruption of system functionality and finally to permanent damage of microelectronic components. Ionizing particles can produce photo-current in active regions of the semiconductor device and lead to phenomena called Single Event Effects. According to 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 [2]" are known as SEUs – Single Event Upsets. SEU is a change of the device state caused by energetic particle passing through the sensitive region of the semiconductor component. High energy particle, through ionization effect, can change the logic state of the data stored into a memory cell, microprocessor register or the FPGA configuration RAM (Random Access Memory) memory. Uncharged particles like photons and neutrons, can induce SEU through an indirect ionization effect [3]. Information stored in the affected logic is lost but component remains functional, therefore error can be corrected if it is detected [4]. Several techniques to protect microelectronic systems against radiation have been developed. Key components of the system can be shielded, unfortunately different type of shield must be used against different type of particles. Electronic components can be hardened during the design and fabrication. Modifications of the critical technological parameters, additional layout processing or modification of integrated circuits electrical construction allow obtaining devices highly immune to soft errors. Unfortunately, this solution significantly increases the production expenses. The hardening-by-system methods like components redundancy, error detection and correction algorithms, memory scrubbing are less expensive and can be used with standard commercial off-the-shelf (COTS) components, but require additional resources and do not guarantee error free operations in the radiation environment [5].

Another solution to make tolerant microprocessor-based systems is a purely programming approach known as the Software-based Radiation Protection (SbRP)[6], [7]. This protection technique enables system to tolerate faults induced by the interaction between radiation and hardware components of the system. When the error occurs, they provide a mechanism to the software to prevent system failure from occurring. They offer services by typically using variables duplication, control sums or redundancy at different levels of system granularity - instruction, blocks of the source code, procedure or entire program. Several specialized protection algorithms adopted to the automatic implementation were developed [8]. To harden data located in a large memory areas like array or structures, Composite Data Type Protection (CDTP) algorithm [9] was formulated. Objects in program are protected by additional set of control sums calculated and checked during every access to variable. To protect local variables used into procedures Live Variable Check (LVC) algorithm [10] was proposed. Every variable in program is duplicated and data consistency is checked at the end of every basic block.

II. Composite Data Type Protection Algorithm

In Composite Data Type Protection each complex variable is treated as set of bytes protected by additional error coding

A. Piotrowski is with the Department of Microelectronics and Computer Science of Technical University of Lodz, Wolczanska 221/223, 90-924 Lodz, Poland, e-mail: komam@dmcs.p.lodz.pl

technique. Every read and write operation performed on the protected object is replaced by the set of operations responsible for checking correctness of variable. In the case of inconsistency, correct value of variable is restored based on redundant information stored together with original data.

Automatic insertion of the CDTP algorithm was implemented in the *cc1* compiler as an independent stage of the compilation process. Protection methods are applied at the beginning of the source code optimization immediately after transformation of program to GIMPLE internal representation. Owing to CDTP algorithm do not introduce any redundancy to the compiled application, every optimization offered by compiler can be used. Modifications of the program are limited only to instruction that perform access to protected variables, therefore techniques like dead code elimination, constant folding or strength reduction can be used to increase speed of application by elimination of unnecessary operation introduced by CDTP algorithm [11].

Algorithm implementation is performed in several, listed below steps:

1) identify every composite data type variables in analyzed source code that are not explicitly marked as non-protected,
2) identify every read and write operations performed on the protected variables,
3) replace every earlier identified variables with appropriate byte arrays. Size of new objects must be sufficient to store original data and additional information about control sum,
4) map multi-dimensional addressing of array elements to one-dimensional offset from the beginning of original variable. If operation cannot be completed during compilation due to lack of necessary information, i.e. array index is calculated during program execution, insert appropriate code to perform mapping in run-time,
5) replace every access to protected object with appropriate accessors functions - for read operation _seu_get_value, for write operation _seu_set_value.

Type of the control sum utilized by the algorithm is not specified, only interface to external library is defined. Example source code before and after implementation of CDTP algorithm expressed in GIMPLE intermediate form is presented respectively in Fig. 1 and Fig. 2. At the beginning, according to the first and second transformation rules, algorithm identifies the *array* variable as composite data type appropriate for protection (Fig 1 line 4) and locates every read and write operation performed on this variable - Fig 1 lines 6, 12 and 27. Original variable is removed and byte array called the *_seu_copy* is introduced (Fig 2 line 3). Size of a new object depends on the selected error coding technique and must be enough to store original data and corrections codes. To set appropriate value of codes, array is first initialized (Fig 2 line 5). After that, all operations performed on the protected variable are replaced by an appropriate function from CDTP protection library. If offset of accessed element can be calculated during compilation appropriate value is used directly (Fig 2 lines 6, 7). On the other hand, if offset must be calculated in run-time, additional source code is introduced - see Fig 2 lines 13 to 15 and 18 to 21.

```
 1  simple_test ()
 2  {
 3
 4   int array[10];
 5
 6   array[0] = 1;
 7   i = 1;
 8   goto <D1876>;
 9   <D1875>:;
10   i.0 = i;
11   D.1879 = i - 1;
12   D.1880 = array[D.1879];
13   D.1881 = D.1880 * 2;
14   array[i.0] = D.1881;
15   i = i + 1;
16   <D1876>:;
17   i.1 = (unsigned int) i;
18   if (i.1 <= 9)
19   {
20       goto <D1875>;
21   }
22   else
23   {
24      goto <D1877>;
25   }
26   <D1877>:;
27   D.1883 = array[9];
28   return D.1883;
29 }
```

Fig. 1. Protected program presented in GIMPLE intermediate representation before CDTP algorithm implementation.

```
 1  simple_test ()
 2  {
 3    unsigned char _seu_copy[80];
 4
 5    _seu_copy = {};
 6    D.1888 = 1;
 7    _seu_set_value (&_seu_copy, 0, &D.1888, &D.1888, 4, 40, 0);
 8    i = 1;
 9    goto <D1876>;
10    <D1875>:;
11    i.0 = i;
12    D.1879 = i - 1;
13    index.3 = D.1879 * 4;
14    f_index.4 = index.3;
15    _seu_get_value (&_seu_copy, f_index.4, &D.1891, &D.1891, 4, 40, 0);
16    D.1880 = D.1891;
17    D.1881 = D.1880 * 2;
18    index.5 = i.0 * 4;
19    f_index.6 = index.5;
20    D.1894 = D.1881;
21    _seu_set_value (&_seu_copy, f_index.6, &D.1894, &D.1894, 4, 40, 0);
22    i = i + 1;
23    <D1876>:;
24    i.1 = (unsigned int) i;
25    if (i.1 <= 9)
26    {
27        goto <D1875>;
28    }
29    else
30    {
31        goto <D1877>;
32    }
33    <D1877>:;
34    _seu_get_value (&_seu_copy, 36, &D.1895, &D.1895, 4, 40, 0);
35    D.1883 = D.1895;
36    return D.1883;
37  }
```

Fig. 2. Protected program presented in GIMPLE intermediate representation after CDTP algorithm implementation.

A. Simulation Results

Four simple programs have been adopted as benchmarks for simulations:

- bubble sort - an implementation of bubble sort algorithm,
- matrix multiplication - multiplication of two matrices,
- FFT - calculation of Fast Fourier Transformation,
- quick sort - an implementation of iterative version of quick sort algorithm.

Additionally, during the tests four type of coding techniques were utilized [12]:

- single-error correct, double-error detect code based on extended Hamming algorithm,
- triple-error correct, quad-error detect code based on extended Golay algorithm,
- full iterated coding scheme,
- selective iterated coding scheme.

In iterated coding scheme, a content of array, independently on the real type of data, is treated as a set of unsigned integer values stored in 2-dimensional matrix, that is protected by additional row and column control sum. In full version of coding scheme, correctness of entire array is checked during every access to variable. In opposite, selective version of algorithm checks only rows and columns involved in current activity. Potentially affected cell is determined based on the following sequence of operations:

- if discrepancy occurs in a row and column checksum, the error affects information located at the intersection of the inconsistent row and column.
- if mismatch is detected only in a row or a column, the error affected the checksum.

Correction of the faulty element can be performed based on the following sequence of operations:

- if error affected the data element of the matrix, it can be corrected by computing XOR operation of the rest of the values on the same row or column and corresponding row or column checksum.
- if mismatch was detected in a checksum variable the correct value can be computed by XOR operation of all elements of the row or column.

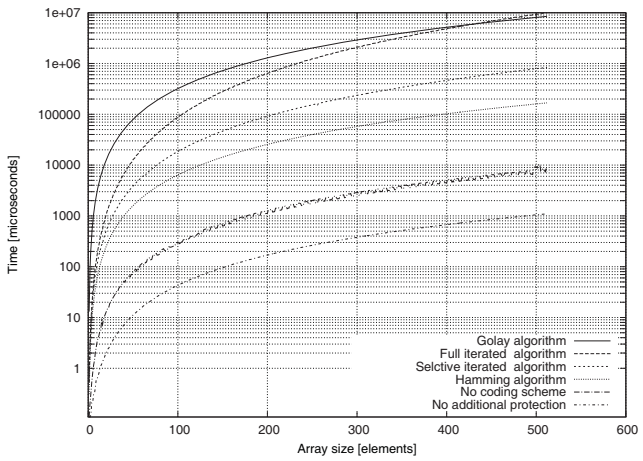

Fig. 3. Performance overhead introduced by CDTP algorithm to bubble sort benchmark application.

Performance overhead introduced by CDTP algorithm to bubble sort benchmark application for different type of coding schemes is depicted in Fig. 3 and summarized for selected array sizes in Table I. Plot labeled *no additional protection* represents time required to finish program by application without implemented CDTP algorithm and curve named *no coding scheme* shows execution time for empty control sum calculation functions. Therefore, difference between both plots

TABLE I
PERFORMANCE OVERHEAD INTRODUCED BY CDTP ALGORITHM TO BUBBLE SORT BENCHMARK APPLICATION.

array size	*no prot. [ms]*	*no coding [ms]*	*golay [ms]*	*hamming [ms]*	*iter. full [ms]*	*iter. sel. [ms]*
10	0,0006	0,0034	3,04	0,0	0,1	0,1
60	0,02	0,1	114,6	2,2	20,3	5,7
110	0,05	0,3	386,7	7,7	113,2	22,8
160	0,11	0,7	824,6	16,3	336,8	54,1
210	0,19	1,4	1421,7	28,1	729,0	102,2
260	0,29	2,1	2180,3	43,1	1376,5	168,3
310	0,40	2,6	3100,8	61,3	2300,5	254,7
360	0,54	3,6	4182,7	82,8	3516,4	361,2
410	0,70	5,3	5425,4	107,7	5138,6	489,8
460	0,89	6,8	6830,4	135,2	7235,0	646,0
510	1,10	7,3	8396,1	166,2	9890,2	827,0

represents application slowdown introduced mainly by replacement of direct access to arrays variable by execution of functions from CDTP library.

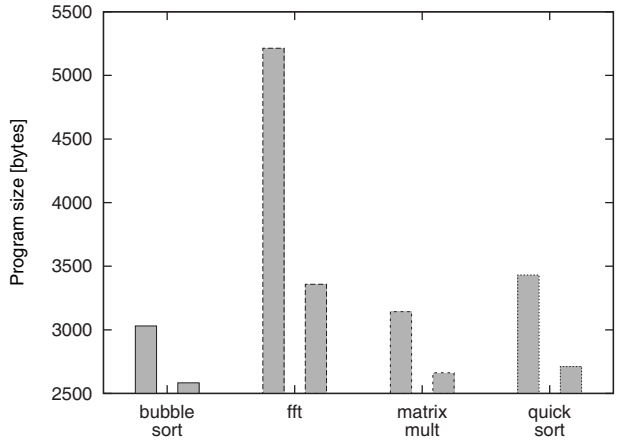

Fig. 4. The size overhead of array consistency check instructions for four test applications. Left bars represent size of application with implemented the CDTP algorithm but with empty error coding library and the right bars represent size of application without CDTP algorithm implementation.

TABLE II
THE OVERHEAD OF ARRAY CONSISTENCY CHECK INSTRUCTIONS.

	bubble sort [bytes]	*FFT [bytes]*	*matrix mult [bytes]*	*quck sort [bytes]*
with protection	3047	5230	3159	3447
without protection	2583	3358	2663	2711
overhead	17,96%	55,74%	18,62%	27,14%

The application size overhead introduced by the CDTP algorithm is determined by the two factors:

- the size of additional code needed for offset calculations and fault detection/correction function calls - depends on application structure and utilization of array variables,
- the size of the error coding algorithm implementation, that is constant for each program.

The application size overhead introduced by instructions responsible for run-time array offset calculations and protection function calls is presented in Fig. 4. Precise results of measurements are stored in Table II. Total size overhead related to

TABLE III
THE SIZE OVERHEAD OF CDTP ALGORITHM.

	bubble sort [bytes]	*FFT [bytes]*	*matrix mult [bytes]*	*quck sort [bytes]*
golay	6302 (144%)	8541 (154%)	6414 (141%)	6702 (147%)
hamming	4647 (80%)	6814 (103%)	4759 (79%)	5031 (85%)
iter. full	5500 (112%)	7532 (124%)	5612 (111%)	5884 (117%)
iter. sel.	5516 (113%)	7564 (125,25%)	5628 (111%)	5916 (118%)
no library	3047 (18%)	5230 (56%)	3159 (19%)	3447 (27%)
no prot.	2583	3358	2663	2711

protection algorithm for different types of benchmark applications and utilized coding scheme is presented in Table III. Row labeled *no prot.* represents size of application without implemented CDTP algorithm and row named *no library* shows total application size in the case of empty control sum calculation library.

III. LIVE VARIABLE CHECK ALGORITHM

Live Variable Check algorithm uses time redundancy to detect computational or memory errors affecting local variables. Secondary set of data is introduced and the same operations are performed on both groups of variables. Duplicated instruction have no effects on result of program, but allows to detect errors in system in run-time. Unfortunately, duplication will cause execution overhead and performance loss. In order to minimize negative effects of algorithm, correctness checking is performed only for variables that are live at the end of processed basic block and were modified within. Results of duplicated computations performed of both copies of variables that are live at the end of basic block, are compared and in the case of data inconsistency, previous state of each variable that is live at the beginning of block is restored and block is reexecuted. Correctness of data must be preserved between multiple basic blocks therefore after positive data verification, variables are stored into so called Recovery Array, protected by Composite Data Type Protection algorithm.

At the beginning of algorithm implementation, non re-executable instructions like array read/write operations or nonpure function calls are located and moved into separate unrepeatable blocks. Next, based on Equations (1) sets of variables that are live before (set IN) and after (set OUT) each basic blocks are calculated. DEF and USE are respectively sets of variables that are defined before use and used before define in block [13].

$$\begin{aligned} IN_B &= USE_B \cup (OUT_B - DEF_B) \\ OUT_B &= \bigcup_{\text{S a successor of B}} IN_S \end{aligned} \tag{1}$$

Each basic block is processed separately according to the following steps:

- at the beginning of basic block a recovery point is established,
- for every scalar variable in a program, secondary variable of the same type is introduced. Every simple instruction is duplicated - secondary statements are inserted immediately after the first one. Function call statements for routines that return value and are marked as *pure*, are duplicated. In the case of *non-pure* functions, located in non re-executable blocks, returned value is simply assigned to the copy of original variable,
- before every operation performed on array located in non re-executable blocks, code that check correctness of index is inserted and in the case of data inconsistency, correct value of variable is loaded from Recovery Array. The use of index value changed by soft error can lead to wrong memory access exception and result in a critical system stop,
- for every variable from *OUT* set modified in processed basic block, data correctness checking instructions are inserted. If *USE* set is not empty, restore basic block is created and values of each variable from USE set are loaded from Recovery Array. Otherwise in the case of fault original basic block can be reexecuted without any load instruction,
- after positive consistency checking and if original basic block ends instruction different then return, new values of live variables modified in block are stored in Recovery Array,
- if block ends with conditional statement, consistency of variable used to determined new control flow directions are checked.

Example source code before and after implementation of LVC algorithm expressed in GIMPLE intermediate form are presented respectively in Fig. 5 and Fig. 6. The control flow graph before and after LVC algorithm implementation are depicted in Fig. 7 and Fig. 8.

```
1  simple_test ()
2  {
3
4    D.1527 = get_int ();
5    if (D.1527 > 10)
6    {
7       t1 = 1;
8    }
9    else
10   {
11     t1 = 2;
12   }
13   D.1528 = t1;
14   return D.1528;
15 }
```

Fig. 5. Source code presented in GIMPLE intermediate form before Live Varible Check algorithm implementation.

A. Simulation Results

Two sets of simple programs have been adopted as benchmarks for simulations. The first one includes:

- bubble sort - an implementation of bubble sort algorithm,
- matrix multiplication - multiplication of two matrices,
- FFT - calculation of Fast Fourier Transformation,
- quick sort - an implementation of iterative version of quick sort algorithm.

The second one includes [14]:

```
simple_test ()
{
  unsigned char _seu_bb [18];

  <L6>:;
   _seu_bb = {};

  <L5>:;
   D.1527 = get_int ();
   __seu_prot_1527.1 = D.1527;
   __seu_store (&_seu_bb, 0, &__seu_prot_1527.1, &D.1527, 4, 18);

  <L4>:;
   if (D.1527 > 10) goto <L8>; else goto <L9>;

  <L7>:;
   __seu_load (&_seu_bb, 0, &__seu_prot_1527.1, &D.1527, 4, 18);
   goto <bb 6> (<L4>);

  <L8>:;
   if (D.1527 != __seu_prot_1527.1) goto <L7>; else goto <L0>;

  <L0>:;
   t1 = 1;
   __seu_prot_t1.2 = 1;
   if (t1 != __seu_prot_t1.2) goto <L0>; else goto <L10>;

  <L10>:;
   __seu_store (&_seu_bb, 4, &__seu_prot_t1.2, &t1, 4, 18);
   goto <bb 5> (<L2>);

  <L9>:;
   if (D.1527 != __seu_prot_1527.1) goto <L7>; else goto <L1>;

  <L1>:;
   t1 = 2;
   __seu_prot_t1.2 = 2;
   if (t1 != __seu_prot_t1.2) goto <L1>; else goto <L11>;

  <L11>:;
   __seu_t_rbb_store (&_seu_bb, 4, &__seu_prot_t1.2, &t1, 4, 18);

  <L2>:;
   D.1528 = t1;
   __seu_prot_1528.3 = __seu_prot_t1.2;
   if (D.1528 != __seu_prot_1528.3) goto <L12>; else goto <L13>;

  <L13>:;
   return D.1528;

  <L12>:;
   __seu_load (&_seu_bb, 4, &__seu_prot_t1.2, &t1, 4, 18);
   goto <bb 5> (<L2>);

}
```

Fig. 6. Source code presented in GIMPLE intermediate form after Live Varible Check algorithm implementation.

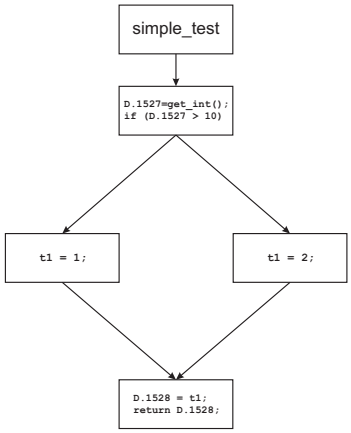

Fig. 7. Control Flow Graph of example application before Live Varible Check algorithm implementation.

- false - an implementation of False Position Method called also *regula falsi* for function root finding,
- ridder - an implementation of Ridders' Method for function root finding,

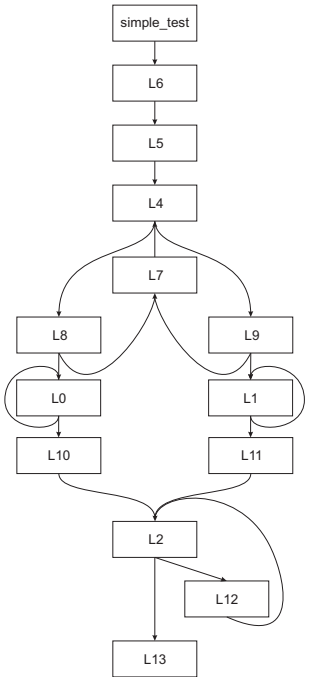

Fig. 8. Control Flow Graph of example application after Live Varible Check algorithm implementation.

- secant - an implementation of Secant Method for function root finding,
- dekker - an implementation of Van Wijngaarden-Dekker-Brent Method for function root finding.

Performance slowdown introduced by LVC algorithm for different type of test application and different level of optimization are presented in Table IV and Table V. Influence of protection method to application size are shown in Table VI and Table VII.

TABLE IV
PERFORMANCE SLOWDOWN INTRODUCED BY LVC ALGORITHM FOR FIRST SET OF BENCHMARK APPLICATION.

	bubble sort [μs]	*FFT* [μs]	*matrix mult* [μs]	*quck sort* [μs]
with protection				
-O0	8489.8 *(11)*	546.1 *(12)*	107601.0 *(9)*	3856.6 *(9)*
-O1	2932.6 *(22)*	41.9 *(10)*	53186.4 *(14)*	2442.5 *(10)*
-O2	2851.8 *(21)*	40.6 *(8)*	49226.6 *(13)*	2316.9 *(16)*
-O3	2820.8 *(21)*	39.3 *(11)*	49987.5 *(13)*	2316.7 *(16)*
-Os	3793.1 *(17)*	40.9 *(6)*	50138.7 *(13)*	2702.6 *(17)*
without protection				
-O0	732.9	46.2	12161.4	433.2
-O1	133.9	4.0	3769.2	244.9
-O2	133.4	5.1	3757.9	140.2
-O3	133.4	3.4	3761.8	141.5
-Os	224.0	6.3	3781.1	161.4

IV. SUMMARY

In this paper, author presents results of simulations involving two Software-base Radiation Protection algorithms: Composite Data Type Protection and Live Variable Check. Both methods are based on redundancy, therefore both introduce additional overhead related to the size of the application

TABLE V
PERFORMANCE SLOWDOWN INTRODUCED BY LVC ALGORITHM FOR SECOND SET OF BENCHMARK APPLICATION.

	false [μs]	*ridder* [μs]	*secant* [μs]	*dekker* [μs]
with protection				
-O0	28.63 *(2,37)*	6.24 *(2,58)*	5.23 *(2,16)*	7.89 *(2,97)*
-O1	31.48 *(2,63)*	6.88 *(2,81)*	5.71 *(2,37)*	11.68 *(4,53)*
-O2	31.30 *(2,61)*	6.93 *(2,86)*	5.58 *(2,38)*	11.71 *(4,56)*
-O3	31.23 *(2,58)*	6.88 *(2,84)*	5.61 *(2,46)*	11.71 *(4,56)*
-Os	27.57 *(2,29)*	5.85 *(2,41)*	5.10 *(2,26)*	8.28 *(3,22)*
without protection				
-O0	12.06	2.42	2.42	2.66
-O1	11.99	2.45	2.43	2.58
-O2	12.01	2.42	2.34	2.57
-O3	12.11	2.42	2.32	2.57
-Os	12.03	2.43	2.26	2.57

TABLE VI
THE AREA OVERHEAD INTRODUCED BY LVC PROTECTION ALGORITHM FOR FIRST SET OF BENCHMARK APPLICATION.

	bubble sort [bytes]	*FFT* [bytes]	*matrix mult* [bytes]	*quck sort* [bytes]
with protection				
-O0	5959 *(2,31)*	19835*(5,91)*	6683 *(2,42)*	8187 *(3,05)*
-O1	4055 *(1,61)*	6742 *(2,38)*	5787 *(2,17)*	5595 *(2,06)*
-O2	4071 *(1,63)*	6214 *(2,22)*	5899 *(2,20)*	5291 *(1,95)*
-O3	4071 *(1,63)*	5318 *(1,91)*	5899 *(2,20)*	5291 *(1,95)*
-Os	3511 *(1,41)*	4579 *(1,67)*	4251 *(1,61)*	4171 *(1,56)*
without protection				
-O0	2583	3358	2763	2683
-O1	2519	2830	2667	2715
-O2	2503	2798	2683	2713
-O3	2505	2782	2683	2713
-Os	2487	2734	2635	2667

TABLE VII
THE AREA OVERHEAD INTRODUCED BY LVC PROTECTION ALGORITHM FOR SECOND SET OF BENCHMARK APPLICATION.

	false [bytes]	*ridder* [bytes]	*secant* [bytes]	*dekker* [bytes]
with protection				
-O0	8320 *(2,57)*	15091 *(3,62)*	6816 *(2,15)*	18368 *(3,90)*
-O1	7167 *(2,27)*	12907 *(3,29)*	6063 *(1,94)*	16527 *(3,76)*
-O2	6735 *(2,11)*	11211 *(2,83)*	5727 *(1,81)*	15215 *(3,40)*
-O3	6735 *(2,11)*	11211 *(2,83)*	5727 *(1,81)*	15215 *(3,40)*
-Os	5332 *(1,69)*	8896 *(2,31)*	4868 *(1,56)*	12196 *(2,85)*
without protection				
-O0	3239	4170	3175	4711
-O1	3163	3923	3131	4395
-O2	3195	3955	3163	4475
-O3	3195	3955	3163	4475
-Os	3163	3843	3115	4283

and time required to finish operation. In the case of CDTP algorithm, it is very important to correctly choose coding scheme utilized by method. Methods with high fault coverage capability, like extended Golay algorithm, are very expensive and introduce large application slowdown. It must be adjusted to expected level of radiation and in consequence number of generated errors. In the case of LVC algorithm, two different sets of benchmark programs were tested. The applications from first set strongly utilized array operations, therefore during method implementation large number of unrepeatable basic blocks is introduced. In most cases, each unrepeatable statement divide basic block into three separate blocks.

REFERENCES

[1] G. Barbottin and A. Vapaille, *Instabilities in Silicon Devices, New Insulators, Devices and Radiation Effects*. North Holland, 1999.

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

[3] R. 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.

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

[5] D. Makowski, "The impact of radiation on electronic devices with the special consideration of neutron and gamma radiation monitoring," Ph.D. dissertation, Technical University of Lodz, 2006.

[6] N. Oh, "Software implemented fault tolerance," Ph.D. dissertation, Stanford University, 2000.

[7] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer Science+Business Media, LLC, 2006.

[8] A. Piotrowski, D. Makowski, G. Jablonski, and A. Napieralski, "The automatic implementation of software implemented hardware fault tolerance algorithms as a radiation-induced soft errors mitigation technique," *Nuclear Science Symposium NSS/MIC/RTSD, 19 - 25 October 2008 Dresden, Germany*, 2008.

[9] A. Piotrowski, D. Makowski, S. Tarnowski, and A. Napieralski, "Automatic implementation of radiation protection algorithms in programs generated by GCC compiler," *European Particle Accelerator Conference, 23-27 June 2008, Genoa, Italy*, 2008.

[10] A. Piotrowski and S. Tarnowski, "Compiler-level implementation of single event upset errors mitigation algorithms," *MIXDES 2009 - Mixed Design of Integrated Circuits and Systems*, 2009.

[11] A. Piotrowski, D. Makowski, G. Jablonski, 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*, 2008.

[12] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*. John Wiley & Sons, 2006.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[14] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical recipes in C: the art of scientific computing*. New York, NY, USA: Cambridge University Press, 1988.

Adam Piotrowski received the MSc and PhD degrees in computer science all from the Technical University of Lodz, in 2004 and 2010 respectively. His research interests include compilation techniques, embedded and fast data acquisition systems. He is involved in development of ATCA-based LLRF control system for XFEL accelerator.