AUTOMATIC IMPLEMENTATION OF RADIATION PROTECTION ALGORITHMS IN PROGRAMS GENERATED BY GCC COMPILER

A. Piotrowski*, D. Makowski, Sz. Tarnowski, A. Napieralski, DMCS, TUL, Lodz, Poland

Abstract

Radiation influence on the microprocessor-based systems is a serious problem especially in places like accelerators and synchrotrons, where sophisticated digital devices operate closely to the radiation source. Reliability of such systems has significantly decreased due to effects like SEU or SEFI. One of the possible solutions to increase the radiation immunity of the microprocessor systems is a strict programming approach known as Software Implemented Hardware Fault Tolerance. SIHFT methods are based on the redundancy of the variables or procedures. Sophisticated algorithms are used to check the correctness of the control flow in the application. Unfortunately, a manual implementation of presented algorithms is difficult and can introduce additional problems with program functionality caused by human errors. Proposed solution is based on the modifications of the source code of the C language compiler. Protection methods are applied at intermediate representation of the compiled source code. This approach makes it possible to use standard optimization algorithms during compilation. In addition, a responsibility for implementing fault tolerant is transferred to the compiler and is transparent for programmers.

INTRODUCTION

The Software Implemented Hardware Fault Tolerance (SIHFT) is a set of algorithms designed to increase reliability and availability of the microprocessor-based systems working in the radiation environment. This is a strict software solution implemented in a high level programming language before the compilation process. According to this paradigm, program not only has to satisfy functional specification, but also has to use special algorithms to monitor functionality, detect, signal and correct hardware errors [1]. SIHFT was designed to protect systems against soft-errors called SEU – Single Event Upset, induced by energetic particles which perform localized ionization events that alter internal data stored in memory or microprocessor registers.

In this paper authors present a new approach to the automatic insertion of SIHFT methods. Several articles and books [2, 3] describe in details theoretical information about software implemented protection algorithms but problem of efficient and error-proof installation of this methods has been always omitted. Manual implementation is a good solution for small projects but is not sufficient for large source code. What is more important, it can be the source of new errors introduced by the programmers.

The new solution is based on automatic implementation of SIHFT algorithms during the compilation process. Several modifications of software methods were proposed to make theoretical algorithms possible to automatic installation. This paper primarily concerns an arrays and pointer-referenced array protection algorithms.

ARRAY PROTECTION ALGORITHM

Arrays are the most common and important composite data types. In most programming language implementations they are stored in a continuous location in memory. Coherence of the whole array variable depends on the coherence of each element separately. Owing to large size of occupied memory, arrays are more sensitive to radiation than simple type variables. For that reasons, independently on the number of dimensions, they are handled as matrices and protected by appropriate row and column checksums based on exclusive disjunction operation. A content of array is treated as a set of unsigned values stored into 2-dimensional matrix, independent on real type of data. Checksum information is kept separately from original matrix into additional variable declared and initialized respectively during the compilation and execution of the program. Memory overhead introduced by the algorithm mainly correspond to the number of elements in the array, see figure 1. Percentage overhead of the occupied memory de-

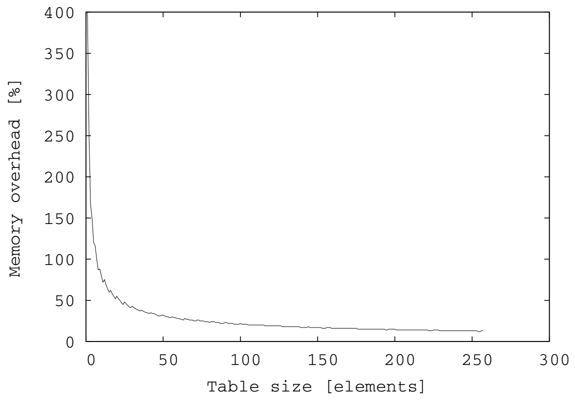

Figure 1: The ratio of number of additional data to an array size expressed in percent in function of array size

creases when the matrix size increases. Automatic installation of array protection algorithm was implemented in the gcc compile as independent stage of the compilation process. Detailed information about proposed solution is presented in [4].

* komam@dmcs.pl

POINTER PROTECTION ALGORITHM

Pointer is a variable which value is a reference to some object. In many C programs it is used instead of the subscripts to iterate over the elements of array. In firmware for embedded systems, pointers are often utilized to get access to the structured data stored in communication buffers. For that reasons, if pointers are used to assure access to protected arrays, additional mechanism must be applied to keep integrity of data. The main problem with pointers is the element it aliases may change over the time. In most cases it is not possible to predict during the compilation process which object will be pointed during the runtime. Constructions like conditional statements or loops introduce ambiguity to the pointer analyses. To solve this problem and protect arrays accessed by the aliases, array protection algorithm is applied however pointed objects are determined not until the execution of the program.

To protect arrays accessed by the pointers several transformations must be introduced to the source code. They are implemented according to the following rules.

- every read and write operation performed on protected array by pointer must be guarded by the array protection algorithm,
- for every pointer that perform access to protected array, additional variable describing aliased object must be introduced,
- for every assignment operation to protected pointer, assignment to variable describing aliases object must be performed. Operation continuity must be saved across the entire program.

Regrettably, rules two and three can be a source of unnecessary operations. Basic version of the protection algorithm requires update of variable describing pointed object for every assign operation. Nevertheless, this is necessary only if assignment to the pointer is performed in parallel with at least two flow graph edges. Optimization presents below solves this problem and allows to decrease size of the final version of application and increase efficiency of the hardened program.

A routine to implement the pointer-referenced array protection algorithm is given in figure 2. The presented procedure performs so-called recursive depth-first search to find every assignment operation for protected variable. The base blocks are visited in the preorder. The data flow for each protected pointer is monitored and analyzed.

Implementation in GCC Compiler

The presented solution was implemented as separate stage of compilation process in the gcc compiler. Additional command line parameters *-seu-protection-pointers-on* and *-seu-protection-pointers-off* were introduced to enable or disable insertion of the protection method.

Pointers, in contrast to arrays, are not protected by default. For that reason programmer must explicitly

```
DirType = enum {indirect, direct}
typeof : Statement → DirType
fdir : DirType
t : Statement
assigns : set of (set of Statement)
procedure ppa (bb : Node, ptr_val : Variable)
begin
    for each statement s ∈ bb do
        if s is an assignment to ptr_val pointer then
            $t := s$
            $fdir :=$ typeof(s)
        if s is an read/write pointed variable then
            protect(s, t, fdir)
    if bb ∈ $DF^+$ then
        if t ∉ assigns[bb] then
            $assigns[bb] := assigns[bb] \cup t$
        if | assigns[bb] | > 1 then
            update(t)
            $fdir := indirect$
    for each $y$ ∈ succ(bb) do
        if | succ(bb) | > 1 then
            push(t)
        ppa(y, ptr_val)
        if | succ(bb) | > 1 then
            $t :=$ pop()
            $fdir :=$ typeof(t)
end
begin
    for each ptr_val ∈ set of protected pointers do
        ppa(ENTRY, ptr_val)
end
```

Figure 2: A routine to implement pointer-referenced array protection algorithm in the intermediate representation of source code

select which variable will be hardened. If particular pointer must be protected by the algorithm, variable has to be defined with the additional attribute called ”*seu_pointer_protection*” i.e. **type_t** *ptr **__attribute** (seu_pointer_protection);. The access to protected array is possible only by hardened pointers. The compiler checks the coherence of the pointer-referenced read and write operations and signal error in the case of inconsistency. For simple test program, shown in listing 1, debugging dump of the intermediate language tree after installation of the protection methods is presented in listing 2. For every protected array, additional structure *__SEU_PROT_TABLE* designed to keep information required by the array protection algorithm is introduced. Each of protected pointers have variable describing aliases object - pointers to a *__SEU_PROT_TABLE* structures. The assignment to this variable is performed according to algorithm presented in figure 2. Functions *__seu_t_crc_cac* and *__seu_t_crc_r* are used to respectively calculate the checksums and, if necessary, correct the array contents, and recalculate the checksums after the array changes. The detail information about the implementation of array protection algorithm is described in [4].

```
int main(){
  int * ptr
  __attribute((seu_pointer_protection));
  int tab_2[32];
  int tab_1[32];
  int D.1544;
 <bb 2>:
  D.1544 = get_int ();
  if (D.1544 > 2) goto <L0> else goto <L5>
 <L5>:;
  ptr = &tab_2;
  goto <bb 5> (<L2>);
 <L0>:;
  tab_1[1] = 7;
  ptr = &tab_1;
 <L2>:;
  *ptr = 1;
  return 0;
}
```

Listing 1: Example of the intermediate representation of source code without implemented pointer-referenced array protection algorithm

```
main(){
  unsigned char __seu_crc_tab_1.1[52];
  struct __SEU_PROT_TABLE __seu_tab_1.2;
  unsigned char __seu_crc_tab_2.3[52];
  struct __SEU_PROT_TABLE __seu_tab_2.4;
  struct __SEU_PROT_TABLE *__seu_ptr.5;

  int * ptr
  __attribute((seu_pointer_protection));
  int tab_2[32];
  int tab_1[32];
  int D.1544;
 <bb 2>:
  D.1544 = get_int ();
  if (D.1544 > 2) goto <L0> else goto <L5>
 <L5>:;
  ptr = &tab_2;
  __seu_prot_ptr.5 = &__seu_prot_tab_2.4;
  goto <bb 5> (<L2>);
 <L0>:;
  tab_1[1] = 7;
  ptr = &tab_1;
  __seu_prot_ptr.5 = &__seu_prot_tab_1.2;
 <L2>:;
  __seu_t_crc_cac(__seu_prot_ptr.5);
  *ptr = 1;
  __seu_t_crc_r(__seu_prot_ptr.5);
  return 0;
}
```

Listing 2: Example of the intermediate representation of source code with implemented pointer-referenced array protection algorithm

Source Code Overhead

The number of additional assignment to the pointer describing aliases object is a function of the program control flow structure and the assignments to protected pointer. In the most pessimistic case, for each assignment additional update of variable describing aliases object is introduced. Let each node *n* in control flow graph have A(n) original assignments to the pointer. The program expands from size $\sum_n A(n)$ to size $\sum_n (A(n) + A'(n))$ where A'(n) is an assignment to the variable describing the pointed object and $A(n) \leq A'(n)$.

CONCLUSIONS

Earlier conducted experiments i.e. [5], proved that the array protection algorithm can be used to increase the reliability and availability of the microprocessor-based systems working in the radiation environment. Pointer-referenced array protection is an extension of array protection designed for automatic insertion during the compilation process. It requires a detailed data and control flow analysis. The main drawbacks of presented methods are increase of a final code size and a decrease of program efficiency. Both disadvantages result from additional operations like comparisons and execution of functions inserted in program to increase reliability of the system. This two methods together form a part of more complex solution for the problem of radiation influence on microprocessor-based systems called Software Implemented Hardware Fault Tolerance.

REFERENCES

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

[2] 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),* page 581, 2003.

[3] O. Goloubeva and M. Rebaudengo and M. Sonza Reorda and M. Violante. Improved software-based processor control-flow errors detection technique. *The Annual Reliability and Maintainability Symposium, Session 14B,* 2005.

[4] A. Piotrowski and D. Makowski and G. Jablonski 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, June 19-21, Poznań (Poland),* 2008.

[5] 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.