EFFICIENT IMPLEMENTATION OF FAULT-TOLERANT DATA STRUCTURES IN PC-BASED CONTROL SOFTWARE

Michael Short
Embedded Systems Laboratory, University of Leicester,University Road, Leicester,UK
mjs61@le.ac.uk

Keywords: Open architecture controllers, software fault tolerance, critical systems.

Abstract: Recent years have seen an increased interest in the use of open-architecture, PC-based controllers for robotic and mechatronic systems. Although such systems give increased flexibility and performance at low unit cost, the use of commercial processors and memory devices can be problematic from a safety perspective as they lack many of the built-in integrity testing features that are typical of more specialised equipment. Previous research has shown that the rate of undetected corruptions in industrial PC memory devices is large enough to be of concern in systems where the correct functioning of equipment is vital. In this paper the mechanisms that may lead to such corruptions and the level of risk is examined. A simple, portable and highly effective software library is also presented in this paper that can reduce the impact of such memory errors. The effectiveness of the library is verified in a small example.

1 INTRODUCTION

Recent years have seen much interest in the use of open-architecture controllers for robotic and mechatronic systems. Such systems typically consist of a combination of commercial off the shelf (COTS) PC equipment alongside motion control, network interface, and sensor/actuator equipment – e.g. (Hong et al. 2001; Short 2003; Lee & Mavroidis 2000; Schofield & Wright, 1998). Such architectures have been used to successfully implement novel control algorithms in a number of research installations (e.g. fuzzy force control (Burn et al. 2003) and H_∞ vibration control (Lee & Mavroidis 2000)); they are also being used in increasing numbers in industrial applications (e.g. KUKA®, STAUBLI®, RWT® systems). The flexibility of such platforms primarily arises by giving engineers the ability to develop and/or modify the control and interfacing hardware and software, which is typically developed in C++.

However, despite increased flexibility and performance (along with marked unit cost reductions), such COTS equipment lacks many of the built-in integrity testing elements which are often employed in more proprietary, specialised control equipment. Many robotic and mechatronic systems, by virtue of their design, are somewhat critical in nature.

A study of available data by Dhillon and Fashandi (1997) concluded that robot-related accidents are primarily caused by unexpected or unplanned motions of the manipulator; a contributory cause of which was malfunctions of the robot control system. Unexpected motions of a manipulator or tooling may result in damage (or complete destruction) of the system itself and any (potentially expensive) equipment in the systems' workspace. Additionally, when considering applications such as surgical robotics, any unexpected or unplanned motion could also result in serious injury or death.

When such COTS equipment is used in situations where their correct functioning is vital, special care must be therefore taken to ensure that the system is both reliable and safe (Storey 1996; Levenson 1995). When considering the equipment employed in a typical robot control system, attention must be paid to potential permanent, transient or intermittent failures of the hardware and software. The need for fault-tolerant techniques is dependant on the potential risk, which is primarily dependant on the application and environment the system is employed in.

Much research has concentrated on providing hardware fault tolerance for such systems (e.g. see Storey 1996). Recent years have also seen the development of several software-based approaches to implementing transient fault detection on COTS processors. They are based around instruction counting, instruction/task duplication and control flow checking (e.g. Rajabzadeh & Miremadi 2006; Rebaudengo et al. 2002; Oh et al. 2000).

Although such techniques are effective at detecting many control flow errors, systems which incorporate them may still be vulnerable to transient errors in data memory (which may not result in control-flow errors). This paper is particularly concerned with the mitigation of transient errors in COTS memory devices used in open-architecture controllers. In section 2 of the paper, we will consider the mechanisms that may lead to memory corruption, the resulting effects, and the level of risk.

In section 3, a simple yet highly effective software library that reduces the impact of memory corruptions and overcomes these implementation difficulties is presented and described in detail. In section 4 we apply this library to a simple test program; a 6 x 6 matrix multiplication program. Fault injection results are described for both the un-hardened and hardened programs. Section 5 concludes the paper.

2 MEMORY ERRORS

2.1 Mechanisms

Corruption of data in memory devices can come from a variety of sources. Single event effects - (SEE's) - caused by particle strikes, may manifest themselves in a variety of ways. They may cause transient disturbances known as single event upsets (SEU's), manifested as random bit-flips in memory. They may also cause permanent stuck-at faults over an array of memory, caused by damage to the read/write circuitry or chip latchup.

In addition, memory devices may also fail due to normal electrical and thermal breakdown effects. Such electrical or thermal failures and disturbances in memory devices may be highly unpredictable, manifesting themselves as complete device failures or stuck-at faults over part (or all) of the memory array.

Memory devices are also susceptible to electromagnetic interference (EMI) from a variety of sources. For example in an industrial robot workcell, numerous devices such as electromechanical relays, motor drives and welding equipment are all sources of noise that are capable of corrupting many electronic circuits (Ong & Pont 2002). Other mechanisms that may lead to memory upsets include power supply fluctuations and radio frequency interference (RFI).

2.2 Level Of Risk

Failure rates for SEU's in ground-based installations are in the region of 10^{-10} - 10^{-12} failures per bit per hour (Normand 1996). Failure rates for individual devices due to electrical effects may be calculated using a methodology such as (MIL 1991); they are typically in the region of 10^{-6} failures per device per hour. Predicting the effects of EMI, RFI and power supply disturbances are extremely difficult and highly dependant on the operating environment and the hardware mitigation techniques that are employed (e.g. signal shielding).

From a practical perspective, experimental studies have demonstrated that on COTS memory devices with built-in integrity checks (such as parity and error correction codes) the problem of undetected memory corruption is large enough to be of concern for some critical systems. Additionally, much PC hardware does not even support such integrity checks (Messer et al. 2001).

For example, a 4MB DRAM memory chip is likely to encounter 6000 undetected memory failures in 10^9 hours of operation (Messer et al. 2001). If a control system PC employs several such devices, with a total of 512 MB memory, this translates to an undetected memory corruption approximately every 55 days of operation.

2.3 Activation Effects

Obviously, not all memory errors will become activated. However, robotic control systems typically involve extremely data-intensive processing with hard real-time constraints. Techniques such as co-ordinate transforms, kinematics, resolved-motion rate control, path planning and force control all typically require hundreds (perhaps even thousands) of matrix manipulations and feedback control calculations every second (e.g. Fu et al. 1997). Considering that a simple 6 x 6 matrix multiplication and storing of the result typically involves 864 memory read/write operations, it can be argued that the probability of activating an error in such control software is relatively high when compared to (for example) a word processing application.

If a memory error does become activated, this can lead to a variety of unpredictable faults (Ong & Pont 2002). For example, they may cause an incorrect value to be output to a port or peripheral;

or they may cause a further area of memory to be corrupted by indexing an array out of its normal bounds. In an open-architecture controller, all of these faults can potentially escalate to full system failures, and cause unpredictable motions of the manipulator or tooling.

2.4 Mitigation Techniques

In order to address this vulnerability, some researchers have investigated the use of Single-Program Multiple Data (SPMD) techniques for data redundancy in both single and multi processor systems (e.g. Redaudengo et al. 2002; Gong et al. 1997). However, such approaches can be problematic from the point of view of the control system developer. Primarily because when the techniques are actually applied, the complexity of the resulting source code can increase dramatically, and the basic meaning of the code can become obscured. This may have an impact on code development, testing and subsequent code maintenance. To illustrate this point, consider the segment of C code shown in Figure 1.

```
#define N (10)
int i;
int a[N],b[N];
 for(i=0;i<N;i++)
   {
   b[i]=a[i];
   }
```

Figure 1: Un hardened code.

For most programmers, this is “self documenting” code, and the meaning is clear (the programmer wishes to copy the contents of any array of ten integers to another array of the same size). Now, consider the same code, hardened using the technique suggested by Redaudengo et al. (2002). This is shown in Figure 2 (note the required checksum initialization code and the XOR macro CHK have been omitted for space reasons). The total code segment, including this initialization (which must be called before each operation), and the CHK macro, is in excess of 36 lines in length; the meaning of the code is also somewhat obscured. In addition, the variable ***i*** in Figure 2 remains un-hardened. If the variable ***i*** were to be hardened, the meaning of the code would become further obscured, with the check-and-correct code for ***i*** embedded within the ***for*** loop construct; as more nested variables are hardened, the problem can soon become difficult to manage. This can be particularly troublesome when writing matrix manipulation code which can often require many levels of nesting.

```
#define N (10)
int i;
int a0[N],b0[N];
int b1[N],b1[N];
int c0,c1;
for (i=0;i<N;i++)
   {
   c0=c0^b0;
   c1=c1^b1;
   b0[i]=a0[i];
   b1[i]=a1[i];
   c0=c0^b0;
   c1=c1^b1;
   if(a0[i]!=a1[i])
      {
      if(CHK(a0,b0)==C0)
         {
         a1[i]=a0[i];
         c1=c0;
         }
      else
         {
         a0[i]=a1[i];
         c0=c1;
         }
      }
   }
```

Figure 2: Hardened code.

Although this problem may be overcome by the use of automatic code generators, this adds an extra level of complexity and abstraction to the software development process, and adds a real possibility of introducing systematic errors into the design process.

In the following section, a software-based methodology will be proposed to simplify the implementation of data redundancy. This technique is an implementation of an SPMD-like architecture to provide fault tolerance to transient errors in data memory. This approach directly addresses the problems of code complexity and compatibility with other software-based approaches. It is primarily suited to C++, but can easily be ported to other object-oriented languages (e.g. JAVA).

3 THE NEW APPROACH

3.1 Requirements

The requirement for this software library was to provide a portable and highly flexible set of new data types for use with C++ programs. The new data types should encapsulate a Triple Modular Redundant (TMR) approach which is completely hidden from the programmer. The data types, to all intents and purposes, appear to the programmer as their basic simplex counterparts; and all the new data types can also be used interchangeably with their simplex counterparts. The library should be as non-intrusive as possible and not require the use of an automatic code generator for its implementation.

Every write operation to the new types invokes a write to three duplicated variables of the corresponding type, and each read operation invokes a two-from-three vote on the duplicated variables. This concept is shown for a generic data type in Figure 3.

We assume that if the data is so corrupted that a two-from-three vote cannot be achieved, then a user-defined error handler is called. This required functionality of this error hander is highly application dependant; it could, for example, freeze the mechanical system and execute a software based self test (SBST) algorithm to verify that no permanent hardware failures have occurred in the CPU peripherals or RAM. Hamdioui et al. (2002) and Sosnowski (2006) have proposed efficient SBST algorithms to achieve this.

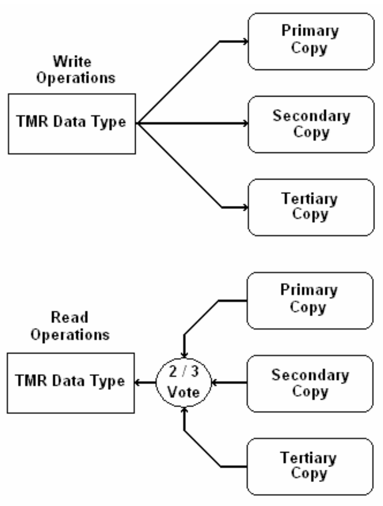

Figure 3: Generic TMR data concept.

3.2 C++ Implementation

In order to create a generic and flexible implementation, the required TMR behaviour was defined in a generic C++ class template named ***TMR_datatype***. The prototype of the class template is shown in Figure 4. This class template for a given data type ***T*** can then be applied to any of the basic in-built C++ data types by means of suitable ***#define*** statements, also shown in Figure 4.

```
template <class T>
class TMR_datatype
{
public:
inline TMR_datatype(const T);
inline TMR_datatype(void);
inline T operator =(const T);
inline operator T();
inline T operator+=(const T);
inline T operator-=(const T);
inline T operator*=(const T);
inline T operator/=(const T);
inline T operator++(int);
inline T operator++(void);
inline T operator--(int);
inline T operator--(void);
inline T operator &=(const T);
inline T operator |=(const T);
inline T operator ^=(const T);
private:
T Primary_Copy;
T Secondary_Copy;
T Tertiary_Copy;
};
#define TMR_int TMR_datatype
<int>
#define TMR_float TMR_datatype
<float>
```

Figure 4: TMR_Datatype class template.

From the class template, it can be seen that each derived object of the template contains three private data declarations, ***Primary_Data***, ***Secondary_Data*** and ***Tertiary_Data***, corresponding to the simplex data type ***T***. The required read and write operations on this data are then achieved by defining new operator member functions using the ***operator*** keyword. It can be seen that all of these operator functions are expanded inline by the compiler, with the use of the ***inline*** keyword; this is to reduce any overheads associated with the call of a member function.

By way of example, the member functions for both the assignment and reference operations on the class template are shown in Figure 5. Note that the use of the explicit reference operator is used in the implementation; thus only the operator functions that explicitly modify the data contents (such as =, ++, --, +=, and so on) needed to be overloaded; this creates a very efficient and portable implementation.

```
inline T TMR_data::operator
=(const T Value)
{
Primary_Copy=Value;
Secondary_Copy=Value;
Tertiary_Copy=Value;
return(Value);
}
inline TMR_data::operator T()
{
if(Primary_Copy==Secondary_Copy)
{
Return(Primary_Copy);
}
else
if(Primary_Copy==Tertiary_Copy)
{
Return(Primary_Copy);
}
else
if(Secondary_Copy==Tertiary_Copy)
{
Return(Secondary_Copy);
}
else
{
Error();
}
}
```

Figure 5: Assignment and reference member functions.

From Figure 5, it can be seen that the TMR behaviour has been captured by the template; when an assignment (write) operator is encountered, the value is written to the three copies of the data. When the reference (read) operator is encountered, a two-from-three vote is employed and the data returned. If no vote is possible, the user defined function ***Error*** is called.

3.3 Hardening Procedure

In Figure 6, the code library described in this section is applied to the code example shown in Figure 1. From Figure 6 it can be seen that the length of the hardened source code is identical to the original and is also highly readable. Additionally, it is noted that – unlike the code shown in Figure 2 - the variable ***i*** is also hardened in this case.

```
#define N (10)
TMR_int i;
TMR_int a[N],b[N];
for (i=0;i<N;i++)
   {
   b[i]=a[i];
   }
```

Figure 6: Hardened code.

From these descriptions it can be seen that this library does not require the use of automatic code generators for its implementation: all that is required is for the programmer to have a basic understanding of the new data types. The hardening procedure can be accomplished extremely rapidly; all that is required is the inclusion of the new TMR template into a project, and altering the variable declarations that require hardening to their redundant counterparts.

4 EXPERIMENTAL RESULTS

To assess the effectiveness of the proposed code library, a fault-injection study was performed on a Intel® Pentium 4-based PC, with a CPU speed of 2.6 GHz and 512 MB RAM, running the Windows NT® operating system. A simple (yet representative) application program was created to perform a 6x6 floating point matrix multiplication. During each experiment, transient faults were injected into the program data area at random times, performing random single bit-flips in the used data areas. The fault injection was performed using a secondary application running on the PC. In the program, the source matrices are first initialized with known constant values. The matrix multiplication is then performed. The values contained in the result matrix are then compared with known constant results. The process then repeats endlessly. Any failures or corrected errors are logged by the application program.

Two different implementations of the program were considered; the normal (simplex) case, and the hardened TMR version. To asses the impact of applying the library on execution time, we also measured the iteration time for each loop of each program using the Pentium performance counter. Table 1 shows the recorded results. In the hardened program, the number of faults injected was increased to reflect the increased size of the program data areas. Fault effects were classified into one of three categories, as follows:

- Effect-less: the fault does not result in a computation failure.
- Corrected: the fault is detected and has been corrected.
- Failure: the fault is not detected or corrected and results in an invalid computation output.

From these results, it can be seen that for both cases, the error activation level was approximately 77%. Application of the TMR data structures increased the execution time of the multiplication task by a factor of 3.2; this is to be expected as we have introduced instruction duplication and voting. Additionally it should be noted that each hardened variable increases the overall memory usage due to its triplicate implementation. The increase in overall program code size was 7.1% in this case.

Table 1: Fault injection results for each program.

	Normal	Hardened
Injected	10000	30000
No Effect	2240	6690
Failures	7760	0
Corrected	0	23010
Calc. Time (us)	4.72	15.27

We can also see from the results that of the 77% of activated faults, 100% of these caused computation failures in the normal program case. The hardened case however, detected and corrected 100% of the activated faults.

5 CONCLUSIONS

In this paper, the mechanisms that can lead to memory corruption in COTS PC control devices have been considered. A novel approach to software implemented fault-tolerance has been presented. The approach, based on an SPMD architecture, can be used to compliment existing error detection and SBST techniques for COTS processors used in open architecture controllers. The approach relies on data and instruction duplication. It has been shown that the method is easily applied, results in readable code, and is able to tolerate 100% of the injected faults in the benchmark described. Whilst the application of the techniques provides high levels of data fault tolerance, there is obviously a trade-off with increases in the code and data size and task execution time. Prospective designers must obviously take these factors into account when considering the techniques.

ACKNOWLEDGEMENTS

The work described in this paper was supported by the Leverhulme Trust (Grant F/00 212/D).

REFERENCES

Burn, K., Short, M., Bicker, R., 2003. Adaptive And Nonlinear Force Control Techniques Applied to Robots Operating in Uncertain Environments. *Journal of Robotic Systems*, Vol. 20, No. 7, pp. 391-400.

Dhillon, B.S., Fashandi, A.R.M., 1997. Safety and reliability assessment techniques in robotics. *Robotica*, Vol. 15, pp. 701-708.

Fu, K.S., Gonzales, R.C., Lee, C.S.G., 1987. *Robotics: Control, Sensing, Vision And Intelligence*. McGraw-Hill International Editions.

Gong, C., Melhem, R., Gupta, R., 1997. On-line error detection through data duplication in distributed memory systems. *Microprocessors and Microsystems,* Vol. 21, pp. 197-209.

Hamdioui, S., van der Goor, A., Rogers, M., 2002. March SS: A Test for All Static Simple RAM Faults. In *Proc. Of the 2002 IEEE Intl. Workshop on Memory Tech., Design and Testing*.

Hong, K.S., Choi, K.H., Kim, J.G., Lee, S., 2001. A PC-based open robot control system: PC-ORC. *Robotics and ComputerIntegrated Manufacturing*, Vol. 17, pp. 355-365.

Lee, C.J., Mavroidis, C., 2000. WinRec V.1: Real-Time Control Software for Windows NT and its Applications. In *Proc. American Control Conf.*, Chicago, Il., pp. 651-655.

Levenson, N.G., 1995. *Safeware: System Safety and Computers*, Reading, M.A., Addison-Wesley.

Messer, A., Bernadat, P., Fu, G., Chen, G., Dimitrijevic, Z., Lie, D., Mannaru, D.D, Riska, A., Milojicic, D., 2001. Susceptibility of Modern Systems and Software to Soft Errors, In *Proc. Int. Conf. on Dependable Sys. And Networks*, Goteburg, Sweden.

MIL-HDBK-217F, 1991. *Military Handbook of Reliability Prediction of Electronic Equipment*. December 1991.

Normand, E., 1996. Single Event Effects in Avionics, *IEEE Trans. on Nuclear Science*, Vol. 43, No. 2.

Oh, N., Shivani, P.P., McCluskey, E.J., 2001. Control Flow Checking by Software Signature. *IEEE Trans. On Reliability*, September 2001.

Ong, H.L.R, Pont, M.J., 2002. The impact of instruction pointer corruption on program flow: a computational modelling study. *Microprocessors and Microsystems*, 25: 409-419.

Rajabzadeh, A., Miremadi, S.G., 2006. Transient detection in COTS processors using software approach, *Microelectronics Reliability*, Vol. 46, pp. 124-133.

Rebaudengo, M., Sonza Reorda, M., Violante, M., 2002. A new approach to software-implemented fault tolerance. In *Proc. IEEE Latin American Test Workshop*, 2002.

Schofield, S., Wright, P., 1998. Open Architecture Controllers for Machine Tools, Part 1: Design Principles. *Trans. ASME Journ. of Manufacturing Sci. & Engineer*, Vol. 120, Pt. 2, pp. 417-424.

Short, M., 2003. *A Generic Controller Architecture for Advanced and Intelligent Robots*. PhD. Thesis, University of Sunderland, UK.

Sosnowski, J., 2006. Software-based self-testing of microprocessors. *Journal of Systems Architecture*, Vol. 52, pp. 257-271.

Storey, N., 1996. *Safety Critical Computer Systems*. Addison Wesley Publishing.