Soft Error Protection via Fault-Resilient Data Representations *

Muhammad M. Latif, Ravi Ramaseshan, Frank Mueller
Dept. of Computer Science, Center for Embedded Systems Research
North Carolina State University
Raleigh, NC 27695-8206, mueller@cs.ncsu.edu

Abstract

Embedded systems are increasingly deployed in harsh environments that their components were not necessarily designed for. As a result, systems may have to sustain transient faults, i.e.*, both single-bit soft errors caused by radiation from space and transient errors caused by lower signal/noise ratio in smaller fabrication sizes. Hardware can protect and even correct transient faults at the cost of redundant circuits. Software approaches can also protect/correct these faults,* e.g.*, by instruction duplication or algorithmic design. Recent work focuses on hybrid solutions of both hardware and software support to counter transient faults while minimizing the cost of protection. While hybrid approaches have been proposed for selectively protecting hardware regions and for control-flow checking, data representations have been widely ignored.*

The contribution of this work is to assess the benefits of inherently error-detecting and optionally error-correcting data representations on the software side. We present some programming patterns which exhibit properties for inherent detection of transient faults. These patterns are compared with techniques which rely on instruction duplication for error detection. Additionally, we introduce a framework to verify the resilience of these patterns with respect to transient faults and compare their performance with other error detection methods.

Preliminary results indicate that a software approach for fault-resilient data representations compares favorably to past work and can reduce the duplication cost in software without compromising error-detection capabilities.

1. Introduction

Transient faults are becoming an increasing concern of system design for two reasons. First, smaller fabrication sizes have resulted in lower signal/noise ratio that more frequently leads to bit-flips in CMOS circuits [5]. Second, embedded systems are increasingly deployed in harsh environments causing soft errors due to lack of protection on the hardware side [25]. The former reason affects computing at large while the latter is predominantly of concern for critical infrastructure. For example, the automotive industry has used temperature-hardened processors for control tasks around the engine block while space missions use radiation-hardened processors to avoid damage from solar radiation.

Current trends indicate an increasing rate of transient faults (*i.e.*, soft errors), not only due to smaller fabs but also because embedded systems are deployed in harsh environments they were not designed for. In commercial aviation, the next-generation planes (Airbus 380 and Boeing 787) will deploy off-the-shelf embedded processors without hardware protection against soft errors. Even though these planes are specifically designed to fly over the North Pole where radiation from space is more intensive due to a thinner atmosphere, target processors lack error detecting/correcting capabilities. Hence, system developers have been asked to consider the effect of single-event upsets (SEUs), *i.e.*, infrequent single bit-flips, in their software design.

In practice, future systems may have to sustain transient faults due to any of the above causes. There exists a significant amount of work on detection of and protection against transient faults. Hardware can protect and even correct transient faults at the cost of redundant circuits [2, 27, 28, 21, 3, 6, 8, 13, 15, 18, 19, 24]. Software approaches can also protect/correct these faults, *e.g.*, by instruction duplication or algorithmic design [20, 14, 11, 10, 23, 12, 4, 9, 7]. Recent work focuses at a hybrid solution of both hardware and software support to counter transient faults [16, 17, 26]. Such hybrid solutions aim at a reduced cost of protection, *i.e.*, cost in terms of extra die size, performance penalty and increased code size. These hybrid solutions are our focus in this research.

Hybrid approaches have been proposed for selectively protecting hardware regions, for control-flow checking and for reduced instruction and data duplication in software [16]. Data representations, however, have been widely ignored.

This work specifically focuses on inherently fault-resilient data representations. Such representations allow the detection of transient faults. We specifically explore the trade-off between performance and code size while preserving the level of reliability.

We identify programming patterns that intrinsically pro-

* This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

vide fault-detecting capabilities within their data representation. We further determine constraints on the context of these patterns and assess the expressiveness of the approach.

The paper is structured as follows. Section 2 gives an overview of previous work on transient fault detection. Section 3 identifies and describes the programming constructs that can be modified for inherent fault detection. Section 4 describes the framework used to test as well as analyze the performance and resilience of the techniques discussed in Section 2. Section 5 shows the performance of the new programming models. Section 6 outlines the future work envisioned. Section 7 summarizes the work.

2. Related Work

Transient faults have been addressed at the level of hardware and software. In the past, IBM has used logic circuits to counter soft errors in its mainframes or by replicating execution units [22], Fujitsu employed error-protection for most latches in the SPARC64 [2] while triple modular redundancy (TMR) was used in the Boeing 777 and a 4-way voting scheme was employed in the Space Shuttle [27, 28, 21]. Other research focused on a variety of hardware schemes [3, 6, 8, 13, 15, 18, 19, 24]. These schemes are generally too expensive for many applications. An alternative is software protection schemes at various levels using checksums, error-correcting code, fault-detection techniques or duplication as well as control-flow checking schemes [20, 14, 10, 23, 12, 4, 9, 7, 26]. Some of these techniques, though mostly implemented in software, rely on additional hardware support.

The most closely related work on software support for reliability is that of error detection by duplicating instructions (EDDI) [11] and the more recent work on software implemented fault tolerance (SWIFT) [16]. Both of these detect soft errors due to a single bit flip (SEU) during the entire program execution.

EDDI provides a pure software solution to the problem assuming no protection on the hardware side. Instructions and program data are duplicated to create a redundant, secondary data flow equivalent to the primary one. Upon a soft error, this equivalence relation is violated. EDDI detects such an error by comparing original and "shadow" values for consistency at "synchronization points", *i.e.*, *prior to write instructions and prior to jumps/branches*. The underlying philosophy is to not let incorrect results leak into the memory system so that after detection of a soft error, a program may be re-executed in part or in full. (The concrete recovery process is beyond the scope of EDDI.) Hence, checks have to precede stores. They are also required prior to jumps/branches since control-flow violations can lead to skipped stores, wrong values for stores or incorrect stores to be executed.

Consistency checking at synchronization points significantly reduces the overhead that would otherwise be incurred after each calculation. The cost of duplication was originally justified by exploiting unused registers and functional units in VLIW machines, such as the Itanium. In embedded systems, the higher memory requirements due to data duplication, the increased register pressure and the lower number of functional units may limit the benefits of EDDI over creating redundant threads of execution.

In addition to instruction duplication, EDDI employs control-flow checking using a basic-block signature scheme. This is necessitated since branches cannot be duplicated, *i.e.*, a single-threaded program can only have a single flow of control. Signature checking on blocks may incur considerable overhead (multiple instructions per block), but this cost can be reduced by hardware support (signature register, checking instructions and a signature transformation generator).

SWIFT reduces the cost of duplication by assuming error protection in the lower memory hierarchy, such as ECC protection in physical memory and some (if not all) levels of the cache hierarchy. The processor core is still unprotected. Hence, SWIFT eliminates the need for data duplication in memory. Instead, data is duplicated at the register level, and equivalence checks as in EDDI are issued prior to writes. Additional hardware support is needed to protect the memory path during read and write operations to avoid soft errors on the bus or within the write buffer [17]. Still, the overhead of instruction duplication and the increased register pressure remain an issue with SWIFT, especially for embedded systems.

3. Design of Fault-Resilient Data Representations

The objective of our work to identify programming patterns that intrinsically provide fault-detecting capabilities within their data representation. The benefit of such techniques lies in the potential to significantly reduce instruction duplication as a means for fault detection. The main challenges of this task are:

1. Program patterns need to be identified that naturally lend themselves to fault resilience.
2. Constraints on the context of such patterns within the application program have to be identified.
3. Compiler transformations have to be developed to automate the transformation step.

We will discuss the first item here while items two and three are discussed in Section 6. In the following, we provide a subset of program patterns to illustrate our approach and present preliminary results to assess code size and performance impacts. The common theme of fault-resilient data-representation is bit-patterns that have self-checking

capabilities in the presence of an SEU, *i.e.*, a *single* bit-flip. For now, our focus is that of fault detection, much in contrast to fault recovery, which will be discussed later.

Fault-resilient conditional encoding is one technique to avoid duplication overhead. As an example, consider the C-like code in Figure 1a. A selection of cases based on an enumeration type for consecutive values is employed, as often found in state-encoding embedded systems.

The code has already been enhanced by EDDI-equivalent fault detection capabilities, as discussed in related work [11]. Each variable is cloned, specifically, x' denotes a copy of the value of x. When selecting a case, a sanity check determines if x or x' has been altered as a side-effect of an SEU, in which case an error is detected. Here, the overhead of EDDI amounts to a duplication of data values and conditionals to check the consistency of original and shadow variables, both of which are operations on the critical path (of regular program execution).

An equivalent, fault-resilient data-representation of x, depicted in Figure 1b, removes the need to duplicate variables and naturally encodes fault detection without any overhead over the original program. Legal values (states) of variable x are encoded as powers of two. Since the hamming distance of any pair of legal values is two, alterations of x by an SEU result in values that are not powers of two. After all, an SEU can only affect one bit, not two.

Fault-resilient loop skewing is a technique that removes duplication overhead for loop iterators. Figure 2a illustrates the overhead of instruction duplication in a loop for EDDI. Figure 2b depicts an equivalent loop for a fault-resilient representation of the iterator. The invariant here is that the iterator be a power of two. Any two iterator values have a hamming distance of two. Furthermore, any iterator value has a checksum of one, which can be checked during loop iteration. To make such checks more efficient, special hardware support for a checksum instruction is required. Notice that the maximum number of iterations is constrained by the word size of the architecture. To generalize this fault-resilient loop pattern to arbitrary number of iterations, *nests of multiple loops* with fault resilience can be employed. Additional hardware support may include support for memory indexing using such counters. Specifically, an instruction can transform a set of counters whose values are powers of two into the equivalent iteration number of the original instruction and, optionally, dereference memory locations with an offset of this index.

Value cloning is another fault-resilient technique. Values are duplicated within a variable whose value range is constraints by at most half the number of bits. Conversely, any short data representation can be replaced with one of twice its size. Consider a 16-bit integer value v. Let v2 be a 32-bit value denoting the value of v in both the 16 least-significant bits (LSBs) and the 16 most-significant bits (MSBs). Some operations on v can be applied to v2 without violation of program semantics, such as the exclusive or operation. Others, such as simple arithmetic (addition, multiplication) require that the operation be executed separately on LSBs/MSBs. Hardware support can provide such instructions that operate on each part in parallel (given two ALUs) or as SIMD instructions (*e.g.*, x86 MMX instructions), particularly when operating on entire arrays.

Fault-resilient pointer traversals are yet another example of reduced duplication overhead. Consider the example in Figure 3a depicting the duplication overhead of the traversed pointer with a consistency check for each loop iteration. A semantically equivalent but fault resilient version is shown in Figure 3b. The size of the data structure has been changed from two to three bytes. This is significant as a multiple of a non-power-of-two number has a hamming distance exceeding one relative to any other multiple of that value. Hence, an SEU can be detected by checking if the offset of the pointer relative to the base of the array is divisible by this number (here: three). However, data structures of odd sizes may adversely affect cache performance due to mis-alignment relative to cache line boundaries. Hence, we use padding so that the resulting structure has an even size. This is particularly easy for larger data structures, *e.g.*, padding a 16-byte structure by 8 bytes. Depending on the cache line size, caches may still be affected, but to a lesser extent, letting this method outperform full duplication as required otherwise.

One other technique that reduces the overhead of duplication is the fault tolerant parameter passing technique. In EDDI, all function arguments are duplicated as shown in Figure 4a. This can introduce a significant overhead if there are frequent calls of *foo*. The fault-tolerant parameter passing technique in Figure 4b tries to reduce this overhead by calculating the hash of all the arguments and then passes the hash as an extra argument into the function. The hash is simply the value of the arguments XOR'ed together. A SEU on any one of the arguments can be detected by the function as the passed XOR value will differ from the runtime XOR of the arguments. In the presence of multiple upsets, the same bit position in different arguments may be flipped, which would remain undetected. Recall that we assume single-event upsets, only, where this method is safe since multi-bit errors have a very low probability.

4. Experimental Framework

We have already designed a fault injection environment catering to data representations for fault resilience. This environment supports native execution with concurrent fault injection *via* a multi-threaded environment, but it is restricted to injection of SEUs in global data, the heap or the stack. Injection into the code segment is currently not sup-

```
enum {A=0,B,C,D};
    //A=0,B=1,C=2,D=3
switch(x) {
  case A: if (x!=x')
    error();
  case B: if (x!=x')
    error();
  ...
}
```

(a) Instruction Duplication

```
enum {A=1,B=2,C=4,D=8};

switch(x) {
  case A:
  case B:
  case C:
  case D:
  default: error();
}
```

(b) Fault Resilience

Figure 1. Encoding for Conditionals

```
for (i=0, i'=0; i<10;
     i++, i'++) {
  if (i != i')
    error();
}
```

(a) Instruction Duplication

```
for (i = 1; i<2^10;
     i = i<<1) {
  if (checksum(i) != 1)
    error();
}
```

(b) Fault Resilience

Figure 2. Encoding for Loops

ported. This environment was utilized to obtain the results in Section 5.

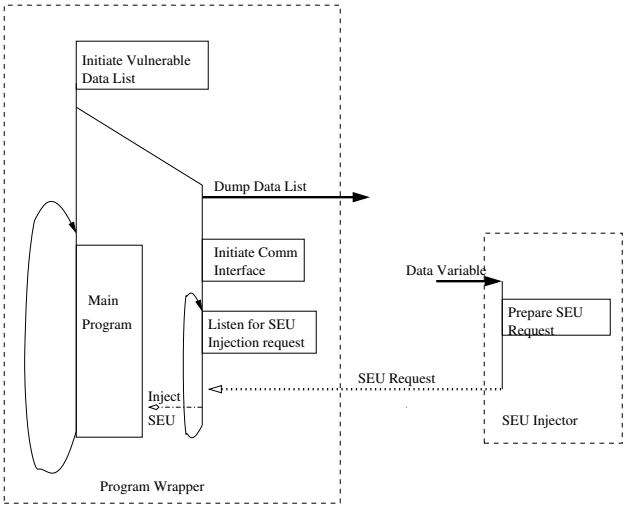

Figure 5. SEU Injection Framework

The framework consists of two different applications: the SEU Injector and the Program Wrapper. The Program Wrapper and the SEU Injector communicate using a socket-based interface. This interface is used to inject an SEU on the data segment in the Program Wrapper. The Program Wrapper uses a set of macros to initiate data structures that store SEU susceptible variables in the form of a variable list. During its initial phase of execution, the Program Wrapper spawns a thread that performs two major operations. It first streams the variable list contents to the program output and then starts the communication protocols to listen for SEU injection request from the SEU Injector application. On receiving an SEU injection request, the thread flips a random bit on requested data variable and loops back to listen for further requests. The main thread, in turn, executes the program logic in an infinite loop. The SEU Injector program is used by passing to it the variable id on which to perform an SEU. A variable id can be selected from the list obtained from the Program Wrapper, as explained earlier. The injector prepares the request and passes it to the Wrapper program via the socket interface.

5. Experimental Results

We have conducted experiments in the fault-injection environment for the examples discussed above: conditional execution, skewed loops, pointer traversal and value cloning as well as parameter checking. Table 1 depicts the additional performance penalty (determined by the timestamp counter hardware register) for (a) full duplication with EDDI and (b) our fault-resilient reduced duplication, both relative to the original program. Notice that the benchmarks *only reflect duplication of the respective construct* while the duplication of other variables and instructions or even control-flow checking is omitted. This allows us to focus on the effect our our technique relative to the equivalent full duplication. As can be seen in the table, EDDI adds performance overhead of 7-100%, depending on the scheme. Our fault-resilient technique inflicts much less overhead. And there is room for improvement. Depending on the probability of

Construct	Conditional	Loop	Pointer	Clone	Parameters
EDDI	7.27%	82.6%	8.14%	56.6%	100.6%
Resilient	4.46%	52.9%	1.19%	26.7%	64.3%

Table 1. Performance Penalty over Original

```
struct {char a,b} st;
st list[100], *pt, *pt';
for (pt=list, pt'=list;
  pt<&list[99];pt++,pt'++){
  if (pt != pt')
    error();
}
```

(a) Instruction Duplication

```
struct {char a, b, buf} st;
st list[100], *pt;
for(pt=list;
  pt<&list[99]; pt++) {
  if(!(pt-list)%sizeof(st))
    error();
}
```

(b) Fault Resilience

Figure 3. Pointer Traversals with Padding

```
call foo(p1,p2,p3,p1',p2',p3')

int foo(int a, int b, int c,
        int a',int b', int c')
{
  int x, x';
  x  = a + b + c;
  x' = a'+ b'+ c';
  if (x != x')
    error();
}
```

(a) Instruction Duplication

```
call foo(p1,p2,p3,p1^p2^p3)

int foo(int a, int b, int c, int XORabc)
{
  int x, x';
  x  = a + b + c;

  if ((x^XORabc) != x^a^b^c)
    error();
}
```

(b) Fault Resilience

Figure 4. Constant Parameters with Checking

an SEU, checks may be moved outside of loop bodies as long as no store is encountered. This technique of code motion will be discussed later.

Table 2 depicts the additional program size for the duplication techniques over the size of the original program. The

Construct	Cond.	For	Pointer	Clone	Parameters
EDDI	15.8%	18.2%	18.9%	19.1%	51.48%
Resilient	5.3%	11.7%	18.9%	15.6%	21.16%

Table 2. Space $[Bytes]$

impact of EDDI on program size can be significant. Embedded systems are often very constrained in memory capacity. Our fault-resilient method consumes significantly less memory than EDDI, combining both better performance with smaller code size.

We finally conducted a test of our methods for a sample program implementing a sorting algorithm of integers. The algorithm was adapted to resemble (a) EDDI duplication without control-flow checking and (b) our data-resilient techniques (no padding). We used level one (O1) and two (O2) optimizations of GCC to ensure that duplication is not removed. Higher optimization levels will remove duplication by invoking common subexpression elimination. The performance results in cycles are depicted in Table 3. The results indicate a relative performance gain of fault-resilient data representations over EDDI that increases (from 13% for O0 to 25% for O1) with higher optimization levels. We expect this trend to continue but could not easily test it in

	EDDI		Fault Resilient	
Optimization	O0	O1	O0	O1
Time	12689	9432	10945	7044

Table 3. Bubblesort $[cycles]$

our current prototyping environment due to compiler limitations. (Duplication will be removed by common subexpression elimination and can be affected by other optimizations that cannot easily be controlled.)

6. Future Work

We are currently developing a systematic approach to transform arbitrary C/C++ programs into fault-resilient ones. This includes efforts to develop static analysis methods that identify patterns suitable for fault-resilient transformation; their implementation in the OpenImpact backend [1]; and development of a hybrid method that meshes our fault-resilient techniques with conventional duplication. We have obtained a copy of the SWIFT enhancements [16] to OpenImpact, which will allow us to directly compare with the EDDI and SWIFT approaches.

7. Conclusion

In this paper, we assessed the benefits of inherently error-detecting and optionally error-correcting data representations on the software side. Programming patterns that exhibit properties for inherent detection of transient faults have been identified and compared with techniques relying on instruction duplication for error detection. We have de-

veloped a framework to verify that the patterns presented and are able to successfully detect an SEU on the programs data segments. The framework has also been used to study the time and space impact of EDDI-like transformations on the patterns discussed in the paper.

Initial results show that inherently error detecting patterns perform better, not just in terms of execution time but also in terms of code size, compared to techniques that use instruction duplication. Our continuing work is currently focusing on the implementation of these patterns in the OpenImpact compiler backend.

References

[1] Openimpact. http://gelato.uiuc.edu/.

[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3ghz fifth generation sparc64 microprocessor. In *Design Automation Conference*, pages 702–705, New York, NY, USA, 2003. ACM Press.

[3] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *International Symposium on Microarchitecture*, pages 196–207, 1999.

[4] G. Chen, M. T. Kandemir, and M. Karaköy. Memory space conscious loop iteration duplication for reliable execution. In *Static Analysis Symposium*, pages 52–69, 2005.

[5] C. Constantinescu. Trends and challenges in vlsi circuits reliability. *IEEE Micro*, pages 14–19, July-August, 1996.

[6] M. Gomaa, C. Scarbrough, T. N. Vijayjumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *International Symposium on Computer Architecture*, pages 98–109, San Diego, CA, May 2003. ACM SIGARCH / IEEE CS. Published as Proc. 30th Ann. Intl Symp. on Computer Architecture (30th ISCA 2003), FCRC'03 ACM Computer Architecture News, volume 31, number 2.

[7] J. S. Hu, F. Li, V. Degalahal, M. T. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction duplication for soft error detection. In *Design, Automation and Test in Europe*, pages 1056–1057, 2005.

[8] A. Mahmood and E. J. McKluskey. Concurrent error detection using watchdog processors - A survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[9] S. H. K. Narayanan, S. W. Son, M. Kandemir, and F. Li. Using loop invariants to fight soft errors in data caches. In *Asia and South Pacific Design Automation Conference*, pages 1317–1320, Shanghai, China, January 18–21, 2005.

[10] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[11] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[12] J. Ohlsson and M. Rimén. Implicit signature checking. In *International Symposium on Fault Tolerant Computing*, pages 218–227, 1995.

[13] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *International Symposium on Microarchitecture*, pages 214–224, 2001.

[14] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 35–44, 2001.

[15] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture*, pages 25–36, 2000.

[16] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture*, pages 148–159, 2005.

[18] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault Tolerant Computing*, pages 84–91, 1999.

[19] N. Saxena and E. McCluskey. Dependable adaptive computing systems – the roar project. In *Intl. Conf. on Systems, Man, and Cybernetics*, pages 2172–2177, Oct. 1998.

[20] P. Shirvani, N. Saxena, and E. McCluskey. Software-implemented edac protection against seus. *IEEE Transactions on Reliability*, 49(1):273–284, 2000.

[21] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

[22] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, Mar./Apr. 1999.

[23] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *International On-Line Testing Symposium*, pages 137–143, 2003.

[24] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *International Symposium on Computer Architecture*, pages 87–98, 2002.

[25] V.Narayanan and Y. Xie. Reliability concerns in embedded system designs. *IEEE Computer magazine*, pages 106–108, January, 2006.

[26] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *International Conference on Embedded Software*, pages 203–209, 2005.

[27] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307, 1996.

[28] Y. C. B. Yeh. Design considerations in boeing 777 fly-by-wire computers. In *IEEE International High-Assurance Systems Engineering Symposium*, page 64, 1998.