Compiler-Assisted Automatic Error Detection

Jan Bessai

Technische Universität Dortmund, Embedded System Software Group,
Otto-Hahn-Str. 16, 44227 Dortmund, Germany
jan.bessai@tu-dortmund.de
http://ess.cs.tu-dortmund.de/Teaching/WS2012/SFt/index.html

Abstract. This work gives an overview of compiler-assisted methods to ensure fault-tolerance without hardware support and major modifications to source programs. The overview of existing methods is accompanied by a practical implementation of AN-Encoding using AspectC++[3], and a theoretical proposal for adding redundancy to functional programs by employing multiple beta reduction strategies. Experimental results from [15, 17] are compared and limitations of compiler-assisted approaches are discussed.

1 Introduction

The increasing ubiquity of computing puts greater and greater importance on reliability of more and more complex systems. In order to ensure that these systems continue to work correctly either better hardware is required or software has to become smarter. Compiler-assisted error detection is a way to automatically enhance software without actually changing its implementation and therefore a very promising area of research. In literature three standard methods are established to have compilers add fault detection mechanisms to programs. These are the MASK approach which is explained in section 2.1, AN-Encoding presented together with a possible implementation as well as variations in section 2.2 and SWIFT outlined in section 2.3. A new approach for functional languages, which has not yet been implemented, is discussed in section 2.4. The next part of this work consists of a short evaluation of existing methods by comparing results obtained in literature (section 3.1) and drawing up general limitations of compiler-assisted error detection (section 3.2). Finally results are summarized and an outlook on future research topics is given.

2 Methods

2.1 MASK

The MASK approach as described in [15] is based on containment of errors, rather than redundancy. Utilizing static analysis compilers can detect which parts of a register are expected to be involved in a computation. This kind of analysis can be performed using the technique shown in [12] where constant

propagation is analyzed bitwise instead of wordwise. All parts of used registers which can be proven to be constant during an operation can be masked out and restored to their previous value if they erroneously change.

```
1 mov ecx, 0
2 and ecx, 0xFFFE
3 again:
4   push ecx
5   call innerLoopFun
6   add ecx, 2
7   and ecx, 0xFFFE
8   cmp ecx, 10
9   jl again
```

Listing 1.1. MASK protected loop counter

Listing 1.1 shows the application of MASK to a loop counter placed in the `ecx` register of a x86 machine. The lowest bit of `ecx` was detected to be constant and is masked out (line 2, 7). Thus the function `innerLoopFun` cannot be called with an uneven value as parameter, even if the last bit of the loop counter flips during addition (line 7). Intelligent compile time analysis with intimate knowledge of the underlying machine model is required as a key factor for MASK to successfully detect and correct errors. This kind of knowledge is hard to maintain on architectures with a diverse instruction set which is constantly under evolution as is the case for x86. Further the compiler has to ensure that commands with indirect register usage do not induce side effects before potential errors are masked out. Such side effects are common in embedded systems where memory mapped registers are often directly tied to output pins of the processor controlling external hardware. If its implementation is feasible, the MASK approach is able to ensure all kinds of invariants, as more complex masking patterns are thinkable. These could range from arithmetic properties as shown before and mentioned in [15] to patterns for valid callable function addresses. Masks on the latter are sometimes even supported in hardware. This is most prominently known from the NX-Bit of the AMD64 architecture [2].

2.2 AN-Encoding

AN-Encoding was invented to protect computers in space travel from harmful effects of background radiation and first employed in the STAR project [1]. It is based on redundancy as it spreads the hamming distance [9, 11] of valid code words. Arithmetic AN-codes are based on multiplication with a large constant commonly denoted as A. Therefore only codewords divisible by A are valid. This property is invariant during addition and multiplication:

$$A \cdot x + A \cdot y \mod A = A \cdot (x + y) \mod A \quad = 0 \tag{1}$$

$$(A \cdot x) \cdot y \mod A = A \cdot (x \cdot y) \mod A \quad = 0 \tag{2}$$

The ratio of all words over valid codewords is reduced from 1 in a non-encoded version to $\frac{1}{A}$ in an encoded version. Thus the error probability for exchanging codewords accidentally is inverse proportional to A. Authors disagree on the choice of A. Both [5] and [15] recommend a value of $2^n - 1$ backed by the observation that any single flipped bit adds or subtracts 2^k which is off by $|(2^n - 1) - 2^k| > 0$ Bits to the next valid codeword. Further efficiency might be improved if multiplication can be implemented by shifting and subtraction ($x \cdot (2^n - 1) = (x << n) - x$). Exceeding this requirement [9] suggests that A should be prime in order to reduce the probability of multiple bit errors adding up as factors of A. Both requirements can be combined using a Mersenne prime [7], for example $2^7 - 1 = 127$. As discussed in 3.1 only single bit error scenarios were experimentally studied and hence the choice should not affect the obtained results.

In contrast to MASK, which requires static analysis capabilities only available within the implementation of a compiler, the aspect of protecting programs by AN-Encoding can be implemented separately. In [9] the compiler back-end of LLVM [13] was adjusted to include appropriate templates. This approach ensures, that all generated code is protected. The source language can also be exchanged since LLVM completely decouples its front- and back-end. Nevertheless the implementation is closely tied to LLVM and its own bytecode assembly language resulting in increased complexity and disadvantageous properties for illustrating how AN-Encoding works. Therefore the following presentation employs a source-to-source transformation, that helps to grasp AN-Encoding at a more abstract level. This comes at the price of not being able to protect compiler generated instructions invisible at the source code level (e.g. pointer arithmetic for address calculations). These instructions are vital to the correct execution of a program, but abnormalities are likely to cause noticeable effects or even crashes, rather than just incorrect computation results. Whether this is acceptable depends on the application context.

For the implementation presented next, C++ was chosen as the source language. No modification to a compiler was needed to add the source-transformation, because AspectC++ [3] provides all the necessary means to weave in the aspect of AN-Encoding. The implementation relies on the lightweight container class `Protected<T>` (listing 1.2), which encapsulates values of its template parameter type `T` to be protected by some form of encoding. It is designed to aid programmers state their intent of protecting a certain value without the need of specifying how the protection is performed. Further it serves as a provider for the necessary join points to weave in advices how to handle encoded computations. All arithmetic operators exist as methods and their assignment versions (e.g. `operator+=`, line 39) dispatch to those methods. If no aspect is present to interfere, `Protected<T>` will just store a value of type `T` (resolved via the generic case of the template definition of `Encoded<T>`, line 4) and mirror the operations defined on `T`. It is almost completely transparent to the programmer using it. The only exceptions are the constructor call, exposed via the function `protect` (line 10), and the cast operation to extract the decoded value. Encapsulating

the constructor is necessary for the mere technical reason, that AspectC++ at the state of writing has no support for construction advices on template classes.

```
1 template <class T>
2 class Encoded {
3  public:
4   typedef T EncodedType;
5 };
6
7 template <class T> class Protected;
8
9 template <class T>
10 Protected<T> protect(const T &value) {
11   return Protected<T>(value);
12 }
13
14 template <class T>
15 class Protected {
16   friend Protected<T> protect<T>(const T&);
17   Protected(const typename Encoded<T>::EncodedType
       &value) : value(value) {}
18  public:
19   typedef typename Encoded<T>::EncodedType EncodedType;
20   EncodedType value;
21   Protected(const Protected<T> &other) :
       value(other.value) {}
22
23   operator T() {
24     return (T)value;
25   }
26
27   Protected<T> operator+(const Protected<T> &y) const {
28     Protected<T> result(*this);
29     result.value += y.value;
30     return result;
31   }
32
33   Protected<T> operator*(const Protected<T> &y) const {
34     Protected<T> result(*this);
35     result.value *= y.value;
36     return result;
37   }
38   ...
39   Protected<T> &operator+=(const Protected<T> &other) {
40     Protected<T> result = (*this) + other;
41     value = result.value;
```

```
42      return *this;
43    }
44    ...
45  };
```

Listing 1.2. Container for encoding protected values

In order to prevent overflows, encoded types need to be four times larger than their unencoded counterparts, since their size has to double for multiplication with A (if A is of the same type) and double again if two encoded values are multiplied. The mapping of types is handled by the class `Encoded<T>` from the previous listing. An aspect header can add partial instantiations of this templated class for each type considered as shown in listing 1.3.

```
1  #include <gmpxx.h>
2
3  template <>
4  class Encoded<char> {
5    typedef int EncodedType;
6  };
7
8  template <>
9  class Encoded<int> {
10   typedef mpz_class EncodedType;
11 };
12
13 ...
```

Listing 1.3. Aspect header, mapping types to their encoded counterparts

For example the encoded type of a 8-Bit char is a 32-Bit int (line 5). The encoded type for a 32-Bit int is an arbitrary precision value (line 10) encoded using `mpz_class` from the GNU Multiple Precision Arithmetic Library [4], because types four times the size of an int are not available in standard C++.

The basic encoding and decoding is realized in the aspect shown in listing 1.4. Here advices on the pointcuts matching calls of `protect` and the cast-operator from `Protected<T>` to `T` multiply with (line 7) and divide by (line 14) the encoding constant A. The decode advice on the cast operator especially avoids problems with calls to foreign libraries which do not accept AN-Encoded values. The compiler will automatically generate calls to cast from `Protected<T>` to `T` when passing protected values to those libraries. Hence in contrast to the method from [9] no further care is required.

```
1  aspect ANEncode {
2    static const int A = 127;
3
4    pointcut encode() =
5      call("% protect<...>(...)");
6    advice encode() : after() {
```

```
7       tjp->result()->value *= A;
8     }
9
10    pointcut decode() =
11      call("% Protected<...>::operator %()")
12      && !negation()
13    advice decode() : after() {
14      *(tjp->result()) /= A;
15    }
16
17    ...
18  };
```

Listing 1.4. Aspect to implement AN-Encoding

Next a variety of different pointcuts is added to match calls to every operator (listing 1.5). Further the advice on the pointcut `checkable()`, defined as the disjunction of all operator pointcuts, ensures the static invariant, that encoded values are divisable by A (line 32). If constraint violations are found the user is currently informed by a simple text message. More sophisticated error reporting and recovery methods could easily be added (e.g. by inheriting from the `ANEncode` aspect).

```
1   pointcut addition() =
2     call("% Protected<...>::operator+(...)");
3   pointcut subtraction() =
4     call("% Protected<...>::operator-(...)");
5   pointcut multiplication() =
6     call("% Protected<...>::operator*(...)");
7   pointcut division() =
8     call ("% Protected<...>::operator/(...)");
9   pointcut negation() =
10    call ("% Protected<...>::operator!(...)");
11  pointcut conjunction() =
12    call("% Protected<...>::operator&&(...)")
13    && args("const Protected<...> &");
14  pointcut disjunction() =
15    call("% Protected<...>::operator||(...)")
16    && args("const Protected<...> &");
17  pointcut exclusiveDisjunction() =
18    call("% Protected<...>::operator^(...)")
19    && args("const Protected<...> &");
20
21  pointcut checkable() =
22    addition()
23    || subtraction()
24    || multiplication()
```

```
    || division()
    || negation()
    || conjunction()
    || disjunction()
    || exclusiveDisjunction();

  advice checkable() : after() {
    if (tjp->result()->value % A != 0) {
      std::cout << "Error detected!" << std::endl;
      std::cout << tjp->result()->value << std::endl;
    }
  }
```

Listing 1.5. Pointcuts matching operators

For multiplication and division additional advice is required to cancel out and supply additional factors of A (listing 1.6).

```
  advice multiplication() : after() {
    tjp->result()->value /= A;
  }

  advice division() : before() {
    tjp->target()->value *= A;
  }
```

Listing 1.6. Additional advice for multiplication and division

Logical operations are harder to encode and have to be replaced by arithmetic versions as described in [9]. This replacement is realized in the advices from listing 1.7. Here casts to `bool` correspond to clipping of values to 0 or 1, where 1 is chosen whenever a value is different from 0.

```
 advice negation() : after() {
    tjp->result()->value = A - A *
        ((bool)tjp->target()->value);
  }

  advice conjunction() : after() {
    tjp->result()->value =
      ((A * ((bool)tjp->target()->value)) *
       (A * ((bool)tjp->arg<0>()->value))) / A;
  }

  advice disjunction() : after() {
    tjp->result()->value =
      ((A * ((bool)tjp->target()->value)) +
       (A * ((bool)tjp->arg<0>()->value))) -
      ((A * ((bool)tjp->target()->value)) *
```

```
16        (A * ((bool)tjp->arg<0>()->value)));
17  }
18
19  advice exclusiveDisjunction() : after() {
20    tjp->result()->value =
21      ((A * ((bool)tjp->target()->value)) +
22       (A * ((bool)tjp->arg<0>()->value))) % 2;
23  }
```

Listing 1.7. Replacement of logic operations

If Bitwise logical operations were implemented via shift and the previous implementation, they would create excessive overhead due to the involvement of loops executing several additions and multiplications per bit. The LLVM based implementation in [9] suggests a lookup table based solution to avoid this. Such lookup tables could either be created by an external codegenerator, or template metaprogramming can be used to create them during compile time. The latter has the advantage that no additional program is required. The basic building blocks for this are shown in listing 1.8 and based on the technique presented in [10].

```
1 template <class T, T ... args>
2 struct TemplateLut {
3   static const T table[sizeof ... (args)];
4 };
5
6 template <class T, T ... args>
7 const T TemplateLut<T, args...>::table[sizeof ...
    (args)] = { args ... };
8
9 template <unsigned int N, class T, template<unsigned
    int, class> class F, T ... args>
10 struct TemplateLutBuilder {
11   typedef typename TemplateLutBuilder<N-1, T, F, F<N,
       T>::value, args...>::result result;
12 };
13
14 template <class T, template<unsigned int, class> class
    F, T ... args>
15 struct TemplateLutBuilder<0, T, F, args...> {
16   typedef TemplateLut<T, args...> result;
17 };
18
19 template <unsigned int N, class T, template<unsigned
    int, class> class F>
20 struct GenerateLut {
21   typedef typename TemplateLutBuilder<N, T, F>::result
       result;
```

```
22 };
```

Listing 1.8. Lookuptables with template metaprogramming

First the struct `TemplateLut` is defined (line 1-4). This struct receives the lookup table type `T` and all table contents as template parameters. To be able to hold arbitrarily many parameters a variadic template argument list is required, which is a feature recently added to C++ [8]. `TemplateLut` includes the final lookup table as the static member variable `table` and initializes it with the contents of the variadic template argument list `args` (line 7). The struct `TemplateLutBuilder` emulates recursive calls to its function like template argument `F` and advances the variadic argument list of the next call by the function result obtained with `F<N, T>::value` (line 11). The template parameter `N` is designated as a counter, which is passed as argument to `F` and decreased in every recursive call. The specialization for `N = 0` in lines 14-17 deals with the recursive base case and passes all function results form its parameter list to `TemplateLut` in order to store them. `GenerateLut` wraps up the recursive logic in an easy to use interface (lines 19-22) where the type `result` will be a specialization of `TemplateLut` that includes a table of `N` results of calls to `F`.

At the time of writing AspectC++ does not yet support the new C++ standard and variadic templates. To work around this separate compilation units can be created, where one unit creates the lookup tables and the other unit references them as pointers to global variables. All required instances have to be known in both compilation units. A preprocessor macro technique described in [6] can be used to create the instantiation code at compile time. Listing 1.9 shows the header file to be shared between both compilation units.

```
1 #ifndef __LUT_H__
2 #define __LUT_H__
3
4 #define LUTS()  \
5   MK_LUT(int, ANNot, 512)
6
7 #ifndef __LUT_IMPL__
8 #define MK_LUT(T, F, N) \
9   extern const T (*F##Table_##T) [N]
10 LUTS();
11 #undef MK_LUT
12 #endif
13
14 #endif
```

Listing 1.9. Shared header for compilation units with and without C++11 features

For compilation units not defining `__LUT_IML__` prior to including the shared header calls to the `MK_LUT(T, F, N)` macro function will create pointers to global lookup tables of type `T` holding `N` elements and name these pointers with the concatenation of the contents of `F`, the text `Table_` and the typename `T` as suffix. For example the definition of `LUTS` in line 5 will create the sourcecode

`extern const int (*ANNotTable_int)[512]`. The compilation unit which includes the code from listing 1.8 has to define `__LUT_IMPL__` prior to inclusion of the shared header and the code from listing 1.10 after `GenerateLut` was defined.

```
1 template <unsigned int A>
2 struct ANNot {
3   template <unsigned int N, class T>
4   struct LutFun {
5     enum { value = /* ... */ };
6   };
7 };
8
9 const unsigned int A = 127;
10
11 #define MK_LUT(T, F, N) \
12   const T (*F##Table_##T)[N] = &GenerateLut<N, T,
       F<A>::LutFun>::result::table
13 LUTS();
14 #undef MK_LUT
```

Listing 1.10. Generator function and macro definition for the C++11 enabled compilation unit

In lines 11-14 of listing 1.10 the macro function `MK_LUT(T, F, N)` is defined again. This time it will instantiate the `GenerateLut` template with its arguments and bind a global variable named as before to the memory location of the created lookup table. The struct `ANNot` provides the function like template struct `LutFun` to create the aforementioned table `ANNotTable_int`. It receives an additional parameter `A` which is designated to be used in `LutFun` for the AN-Encoding of tabulated bitwise logical negation. The concrete definition of this function is omitted. While tests have shown that the presented mechanism can correctly compute tables for any defined function, at the time of writing no useful definition for a function to create tables for bitwise logical negation is known to the author. In [9], where the proposal to precompute results was made, only a vague description how this can be done is given. All trials to reproduce it programmatically or by hand failed. Nonetheless the missing piece should be easy to insert once it is found, and all important functionality to implement a source to source transformation aspect was presented.

AN-Encoding in its presented form is agnostic to which parameters are passed in, as long as they are encoded as multiples of A. This leaves a window of vulnerability for errors causing exchanged operands. In [16, 17] two extensions are proposed. The first, called ANB-Encoding, adds designated constants for

every parameter to the operation result value. The basic operations then become

$$\begin{aligned} A \cdot x + A \cdot y + B_x + B_y \mod A &= \\ A \cdot (x + y) + B_x + B_y \mod A &= B_x + B_y \qquad (3) \\ (A \cdot x) \cdot y + B_x + B_y \mod A &= \\ A \cdot (x \cdot y) + B_x + B_y \mod A &= B_x + B_y \qquad (4) \end{aligned}$$

where B_x and B_y have to be chosen differently for each operational parameter. A wrong parameter, e.g. B_z, can then be detected by an offset to the expected modulo result. The second proposed extension, ANBD-Encoding, deals with control flow errors. A new offset D is added to each operation:

$$\begin{aligned} A \cdot x + A \cdot y + B_x + B_y + D \mod A &= \\ A \cdot (x + y) + B_x + B_y + D \mod A &= B_x + B_y + D \qquad (5) \\ (A \cdot x) \cdot y + B_x + B_y + D \mod A &= \\ A \cdot (x \cdot y) + B_x + B_y + D \mod A &= B_x + B_y + D \qquad (6) \end{aligned}$$

If an operation is executed out of order, the current value for D - kept and updated in a register - will again cause a detectable difference to the expected modulo result. Of course this kind of protection is limited in its capability find errors where exchanged operands and out of order execution accidentally add up to satisfy the required invariant.

2.3 SWIFT

Perhaps the most intuitive way to detect and recover errors is to run instructions more than once and compare the outcomes. This approach is called SWIFT and is studied in [15]. Listing 1.11 shows x86-Assembly code, in which an `add` instruction (lines 2 and 5) is executed twice and its results are compared calling an error handler on mismatch.

```
1 push eax
2 add eax, ebx
3 push eax
4 mov eax, [esp + 4]
5 add eax, ebx
6 push ebx
7 mov ebx, [esp + 4]
8 cmp eax, ebx
9 jeq ok
10 call errorHandler
11 ok: ...
```

Listing 1.11. SWIFT protected add

If the instruction is executed three or more times majority voting can be employed to decide which result was correct [15]. As it can be easily seen in listing

1.11 not only the extra cost of repetitive instructions will arise, but especially on register starved architectures like x86 additional commands will be needed to save intermediate results on the stack. In theory these commands of course can fail again and might need protection. Therefore the use of SWIFT, also it is simple to implement, has to be carefully justified [17]. Another obstacle, similar to MASK, is to ensure the absence of repeated side effects [17]. Thus again intimate knowledge of the underlying instruction semantic is needed. In contrast to AN-Encoding protection from permanent failures or burst failures is impossible with SWIFT [17]. If repeated instructions all calculate the same wrong result, no error will be detected.

2.4 Multi Strategy Beta Reduction

The last section described the SWIFT approach on a very low level for imperative machine instructions. If the underlying target language is not imperative, but functional, compilers can employ a similar scheme. In the lambda calculus, which is the fundamental base of all functional languages, program evaluation is done by beta reduction, usually defined as

$$\begin{aligned}
(\lambda x.P)Q &\rightarrow_\beta P[x/Q] \\
x &\rightarrow_\beta x \\
P \rightarrow_\beta P' &\Rightarrow \lambda x.P \rightarrow_\beta \lambda x.P' \\
P \rightarrow_\beta P' &\Rightarrow P\,Q \rightarrow_\beta P'\,Q \\
Q \rightarrow_\beta Q' &\Rightarrow P\,Q \rightarrow_\beta P\,Q'
\end{aligned}$$

where $P[x/Q]$ denotes the substitution of x by Q in P as defined in [18]. The last two rules for beta reduction show, that the application of the term P to Q, where $P \rightarrow_\beta P'$ and $Q \rightarrow_\beta Q'$ can be reduced to either $P'\,Q$ or $P\,Q'$. By the Church-Rosser-Property [18] for two different reductions with the same starting point there always exists a finite reduction to a common form. The next example shows how the same lambda term is reduced using two different reduction strategies ($\rightarrow_{\beta,1}$ and $\rightarrow_{\beta,2}$).

$$\begin{aligned}
&(\lambda x.x((\lambda y.y3)x)))f \\
\rightarrow_{\beta,1} &(\lambda x.x(x3))f \\
\rightarrow_{\beta,1} &f(f3) \\
&(\lambda x.x((\lambda y.y3)x)))f \\
\rightarrow_{\beta,2} &f((\lambda y.y3)f)) \\
\rightarrow_{\beta,2} &f(f3)
\end{aligned}$$

Theoretically a compiler could make use of this to create redundant different control flows which can be evaluated to a point where they converge to a common result. Many type systems like the simple typed lambda calculus $\lambda^\rightarrow$ even provide strong normalization properties, such that any reduction can be chosen

and a normal form can be found [18]. This variation of SWIFT would be well suited to deal with burst failures as these are unlikely to occur in two different control flows. Further pure functional programs are free of side effects by definition, so no harm is done by executing the same function twice. Nevertheless the runtime system employed in real world functional compilers [14] would need to be protected from failures itself, for which only other methods are suited.

3 Evaluation

3.1 Experiments

Testing the efficiency of measures for fault tolerance is a difficult task. The main aspects which have to be taken into consideration are the fault model used to inject errors, the programs under test and whether and how performance is measured. The situation is worsened by the fact, that errors occur randomly with random effects, and a large quantity of experiments has to be conducted in order to generalize results. All these reasons made it impossible to run own experiments within the scope of this work. They also reflect heavily on the generality of previous results. Currently the most complete studies on the topic are [15] and [17]. Other papers usually provide only summarized results of single methods. In [15] only one error per program could appear and only single bit errors were examined. In [17] faulty operands, faulty operators, lost operations, permanent failures and burst errors were studied as well. Programs under test were in both cases a range of different standard programs, where the test suite in [15] was much more diverse. The tested fault prevention mechanisms were in both studies a range of variations and combinations of those explained here. In [15] special attention was payed to the combination of different approaches, whereas in [17] the main goal was to compare different versions of AN-Encoding to SWIFT. Performance was commonly measured in terms of undetected errors (Silent Data Corruption, SDC). Nevertheless [15] looked at errors which do not affect the application result and errors leading to crashes as well. Further in [17] throughput in a client-server scenario was observed while in [15] runtimes were compared to unprotected versions. Both studies employed large quantities of runs, where in [15] 250 runs were executed per benchmark while in [17] several thousand runs were executed per injected error type.

The differences between both studies are too great to rank approaches. A common result also shared in [9] and [16] is, that by the use of appropriate compiler based error detection methods undetected error rates can be brought down by about 80-95%, while the runtime performance of programs is degraded by a factor of roughly 2-3.

3.2 Limitations

Several specific limitations of compiler based approaches were already discussed during the presentations in the previous sections. In addition to these it is important to mention a few general limitations that apply to all compiler based error

detection mechanisms. One major limitation of all software based ways to ensure fault tolerance is, that always additional instructions are required. Those instructions can fail and cause errors instead of preventing them. They are especially error prone if the memory in which they are stored is faulty. The unreliability introduced by overhead can be further worsened if protection mechanisms lack granularity and protect uncritical sections of programs. The aspect oriented implementation of AN-Encoding presented in section 2.2 is one step towards better granularity as it allows the programmer to specify which computations are to be protected. Another very general limitation is that compilers usually are unaware of the semantic of the source language of a program and can only decide based on very coarse heuristics which protection mechanism is best. The approaches in all research papers known to the author don't even try to capture the semantic of the source program at all. Instead they focus on low level target languages, which are very sensitive to subtle details of specific implementations. Further, even though it is possible, no approach explicitly dealt with failover cleanup and provided programmer controlled interfaces for error scenarios. This is mainly due to the lack of production grade standardized implementations.

4 Summary and Outlook

Four methods for compiler-assisted error detection were presented. The MASK approach discussed in section 2.1 is based on error containment and relies heavily on the static analysis capabilities of compilers. It has a very small instruction footprint but can be hard to implement. AN-Encoding presented in section 2.2 employs value level redundancy. Its principles are easy to understand and it is capable of protecting a broad variety of operations against different kinds of errors. A very lightweight implementation was sketched out, which uses AspectC++ to weave AN-Encoding into the sourcecode of a program. The advantages and limitations of this style of implementation were discussed and compared to the compiler back-end implementation presented in [9]. The SWIFT approach adds redundancy at an operational level and was outlined in section 2.3. Even though it is very intuitive there are pitfalls like side effects, which are to be considered before it is applied. The multi strategy beta reduction approach (section 2.4) is a new proposal from this work and similar to SWIFT, but for functional languages. In theory it is capable of adding different redundant execution paths with equal results to a program. A comparison of experimental results (section 3.1) published in literature highlighted the many possibilities arising when testing the effectiveness of the presented methods. While this made it impossible to fairly rank mechanisms, it revealed that their correct use can drastically lower the probability of undetected errors. The final discussion of general limitations to compiler-assisted error detection showed that a lot of work has yet to be done. Interesting further research topics are the incorporation of source language semantic into error detection mechanisms, the standardization and maturation of compilers employing any of the presented methods and to find out in how far the multi strategy beta reduction approach is feasible to implement.

Bibliography

[1] Computers in Spaceflight: The NASA Experience (1987), http://history.nasa.gov/computers/contents.html

[2] AMD64 Architecture Programmer's Manual (2012), http://support.amd.com/us/Embedded_TechDocs/24593.pdf

[3] Homepage of the AspectC++ language (2013), http://www.aspectc.org/

[4] The GNU Multiple Precision Arithmetic Library (2013), http://www.gmplib.org

[5] Avizienis, A.: Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. IEEE Transactions on Computers C-20(11), 322 – 1331 (Nov 1971)

[6] Bright, W.: The X Macro. Dr. Dobb's Bloggers (2010), http://www.drdobbs.com/cpp/the-x-macro/228700289#

[7] Caldwell, C.K.: Mersenne Primes: History, Theorems and Lists (2013), http://primes.utm.edu/mersenne/

[8] Du Toit, S., et al.: Working Draft, Standard for Programming Language C++. Tech. rep., ISO/IEC (2012), http://isocpp.org/std/the-standard

[9] Fetzer, C., Schiffel, U., Skraut, M.: An-encoding compiler: Building safety-critical systems with commodity hardware. In: Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol. 5775, pp. 283–296. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04468-7_23

[10] Fritzsche, G.: Programmatically create static arrays at compile time in C++ (2013), http://stackoverflow.com/a/2981617

[11] Haming, R.W.: Error Detecting and Error Correcting Codes. The Bell System Technical Journal 26, 147–160 (1950)

[12] Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 194–206. POPL '73, ACM, New York, NY, USA (1973), http://doi.acm.org/10.1145/512927.512945

[13] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)

[14] Marlow, S., Peyton Jones, S.L.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. Journal of Functional Programming 16(4-5), 415–449 (2006)

[15] Reis, G.A., Chang, J., August, D.I.: Automatic Instruction-level Software-Only Recovery. IEEE Micro 27, 36–47 (2007), http://doi.ieeecomputersociety.org/10.1109/MM.2007.4

[16] Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In: Schoitsch, E. (ed.)

Computer Safety, Reliability, and Security. Lecture Notes in Computer Science, vol. 6351, pp. 169–182. Springer Berlin / Heidelberg (2010)
[17] Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: Software-Implemented Hardware Error Detection: Costs and Gains. In: The Third International Conference on Dependability (DEPEND 2010). pp. 51–57. IEEE Computer Society, Los Alamitos, CA, USA (2010)
[18] Sørensen, M.H.B., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism (1998), http://ls14-www.cs.tu-dortmund.de/images/d/db/Curry-howard.pdf