Unreliable yet Useful – Reliability Annotations for Data in Cyber-Physical Systems

Michael Engel, Florian Schmoll, Andreas Heinig, and Peter Marwedel
Design Automation for Embedded Systems
Faculty of Computer Science
TU Dortmund, Germany
Email: {firstname.lastname}@tu-dortmund.de

Abstract: Today, cyber-physical systems face yet another challenge in addition to the traditional constraints in energy, computing power, or memory. Shrinking semiconductor structure sizes and supply voltages imply that the number of errors that manifest themselves in a system will rise significantly. Most cyber-physical systems have to survive errors, but many systems do not have sufficient resources to correct all errors that show up. Thus, it is important to spend the available resources on handling errors with the most critical impact.

We propose an "unreliability" annotation for data types in C programs that indicates if an error showing up in a specific variable or data structure will possibly cause a severe problem like a program crash or might only show rather negligible effects, e.g., a discolored pixel in video decoding. This classification of data is supported by static analysis methods that verify if the value contained in a variable marked as unreliable does not end up as part of a critical operation, e.g., an array index or loop termination condition. This classification enables several approaches to flexible error handling. For example, a cyber-physical system designer might choose to selectively safeguard variables marked as reliable or to employ memories with different reliability properties to store the respective values.

1 Introduction

Ongoing advances in fabrication technology and new demands on energy conservation require the use of continually shrinking semiconductor sizes and supply voltages in processors, peripherals, and memories for cyber-physical systems. These reductions, however, also bring along some undesirable consequences. The most dire consequence is that the number of *transient faults*, caused by natural radioactive decay, cosmic particles like neutrons, or disturbance in supply voltages, will be increasing significantly, as predicted by the International Technology Roadmap for Semiconductors [ITR].

The common approach to build systems tolerant to transient faults is to introduce some sort of *redundancy* into a system, like additional Error Check and Correction (ECC) bits and control logic safeguarding main memory, register, or cache contents, or Triple Modular

This work is supported by the DFG Priority Program SPP1500 under grant no MA943/10-1

Redundancy (TMR), which triplicates all code execution in a system and uses a voting mechanism to detect incorrectly calculated results.

Cyber-physical systems, however, often cannot afford to spend too many resources on redundancy. Consequently, typical cyber-physical systems will be victim to an increased rate of system failures. A novel approach is thus required to increase the likelihood that a system survives the effects of a transient fault.

Our resource-conserving approach to fault tolerance applies a *classification* to the occuring faults in order to assess the worst possible outcome of a fault. For faults affecting main memory, we have already applied this method successfully to an H.264 video decoder [HESM10b]. In this application, certain faults lead to insignificant effects like a discolored pixel on the screen or even no effect at all, while other faults may cause the application to crash or cause severe image distortions. Accordingly, priority is given to the correction of errors with the most significant impact.

However, it is still an open question which information should be provided to a system in order to support the classification. One possible approach is to selectively correct errors only in certain code sections (e.g., specific functions) of an application [HESM10a]. However, certain pieces of data traverse through multiple functions of an application. A fault affecting the integrity of this data in any location in the related code flow might have a fatal consequence, since, e.g., this data finally will end up being added to a base pointer. An incorrect value added to a pointer may lead to any number of undesirable fatal consequences from overwriting some unrelated value in memory to accessing an unmapped region of memory, possibly leading to a system crash.

Thus, in this paper we propose an approach to annotate requirements to variables and data structures in an application using *reliable* and *unreliable* identifiers. Unexpected modifications of variable values marked as reliable may possibly have dire consequences, whereas transient faults affecting an unreliable variable will only have minor consequences. To apply such a classification, a large amount of *application knowledge* is required (typically provided by the application's developers), since it must be known where a value might end up while considering possible side-effects. However, not all of this required application knowledge is readily available to create classification annotations that mark a specific variable as reliable or unreliable. Thus, we propose an aid for the annotation process that applies *static analysis* methods to the source code to identify candidates that might be marked as unreliable.

The rest of this paper is organized as follows. Section 2 describes the idea behind reliability annotations, followed by a discussion of applicable data flow analyses to support the annotations in section 3. Section 4 describes our first prototypical implementation and an evaluation based on a JPEG image decoder. Section 5 discusses related work, and section 6 concludes the paper and gives an outlook to future work.

2 Reliability Annotations

In traditional fault-tolerant systems, every error showing up is handled in the same way, i.e., there is no distinction between errors according to their impact. In many cyber-physical systems, however, providing methods to correct every error will exceed the available resources. For example, in a real-time system, applying error correction might violate the timing constraints if a processor is used that is just fast enough to handle the task. Thus, conventional fault-tolerant systems are often rather overspecified.

If a sufficient amount of resources is not available, a system should at least provide a *best-effort* approach to handle errors. In practice, this means that the most critical errors showing up in a system, like program crashes, should be corrected while errors with a less significant impact, like a pixel of a wrong color that is displayed on the screen of a video player, will be ignored. Obviously, the definition of a *critical* error is application-dependent. A green instead of a red pixel in a video frame will in most cases not be perceived, whereas a green instead of a red light shown on traffic lights can have more dire consequences. Thus, in order to distinguish between critical and less critical errors, application knowledge has to be provided in the form of annotations. This knowledge is then used to provide a classification of faults according to their impact.

In the following sections, we describe an approach that provides annotations that indicate if the data contained in a variable or data structure is expected to be **reliable** – i.e., faults in the specific item cannot be tolerated – or **unreliable**, indicating that a modification to that specific item will have non-fatal consequences.

2.1 Reliability-aware Data Types

We propose to provide the annotations in the form of a type qualifier that explicitly expresses reliability requirements. These requirements can on the one hand express that a piece of data is critical for the dependability of a system, thus the demand for reliability has to be marked using a *reliable* qualifier (e.g., a `reliable int`).

On the other hand, to reduce the resource requirements of error handling, it is desirable to explicitly mark data that only cause minor disturbance in case of a transient fault using the *unreliable* qualifier. If a fault affects a data item marked as unreliable, error correction is not required. Nevertheless, if sufficient resources are available, an error in such a piece of data may be corrected in order to improve a system's quality of service. In the absence of errors, data marked as unreliable behaves like the regular, non-qualified data type.

2.2 Type Synthesis for Reliability Annotations

Marking a variable or data structure as unreliable alone is not sufficient to ensure that unreliable data has no effect on pieces of code that may lead to critical errors. Thus, the unreliability notion of a data element has to be propagated.

If an expression in a program's source code contains at least one unreliable data type, the complete expression is *tainted*. The result of such an expression may not be automatically assigned to a reliable data type; instead, a compiler has to notify an error. Thus, an expression like

```
unreliable int u; reliable int r; r = u + 2;
```

will cause a compiler error, whereas

```
unreliable int u; reliable int r; u = r + 2;
```

is a legal construct.

2.3 Assigning Unreliable Data

In some cases it may actually be desirable to assign the result of an operation containing an unreliable data type to a reliable data element. If required, this – potentially dangerous – assignment can be realized by using an *explicit typecast* to a reliable data type, such as:

```
unreliable int u; reliable int r; r = (reliable int)(u + 2);
```

Using this type cast, the programmer explicitly acknowledges that precision may be lost in this assignment.

2.4 Unannotated Data

If every piece of data in use by an application would have to be annotated, the overhead for the developer would be significant. However, it is not obvious how to handle non-annotated data in case of a fault. In fact, this behavior should be selectable by the developer.

2.5 Application of Annotations

Providing the annotations is only the first step to support a flexible, classification-based approach to error handling. Additionally, the system developers have to provide a separate

implementation for the annotation-based *fault tolerance*. In this paragraph, we describe several possible approaches to fault-tolerance methods that can profit from reliability annotations. We expect that several of these approaches have to be combined in order to create a consistently reliable system.

Memory hierarchy In a system using several types of memory (e.g., caches, scratchpads and regular DRAM), like shown in Figure 1, a system designer may decide to selectively protect certain elements of this memory hierarchy. One example for this would be a system using an ECC-protected scratch pad memory, whereas for cost reasons, the larger main memory is unprotected against transient memory faults.

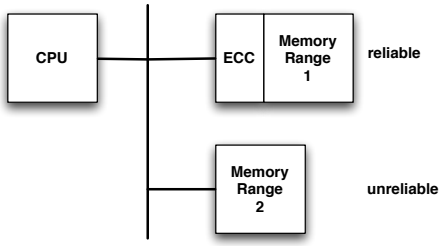


Figure 1: Memories with different reliability

According to the annotations, the compiler is instructed to place reliable and unreliable data in different sections. An appropriate linker script can then take care of mapping reliable data objects to an ECC-protected memory space, whereas objects marked unreliable will be mapped to non-protected memory.

This approach, however, will only work for global objects without extensive modifications of the compiler's memory management. Since function-local variables are expected to reside in contiguous regions on the system's stack, the stack will have to reside in reliable memory. A multi-stack setup could enable a compiler to allocate local data marked as unreliable on a separate stack in unprotected memory. However, such a setup would require considerable changes to the compiler backend and ABI for the target system. A similar approach has been proposed in [YPPJ06] to protect a system against buffer overflows.

In addition, this solution will only protect data from faults in main memory. To build a comprehensive fault-protection solution, additional parts of the system's memory, like registers and caches, would need to be protected against transient faults.

Mapping to multicores In multicore systems, a parallelizable application is usually mapped to the available cores in order to speed up its average or worst case execution time. A multicore system may, however, also contain cores with different reliabilities – e.g., some cores might be running on lower supply voltages in order to conserve energy and are, thus, more susceptible to transient faults. Using a parallelization approach driven by reliability annotations, pieces of code operating on unreliable variables only could be

explicitly mapped to unreliable cores, thus improving the overall reliability of the application.

A similar mapping approach on a task level, providing certain tasks of a system with dual modular redundancy, whereas other high-performance tasks run without redundancy, was proposed in [WCS09]. However, that system only provided task-level granularity, which may be too coarse-grained for our approach.

Redundant execution Yet another way to reduce the overhead of fault-tolerance methods is selective redundant execution. Systems performing redundant executions perform the same operation multiple times, often using different hardware resources, e.g., different source and destination registers. The results obtained are compared to each other; if the results differ, the operation is considered to be affected by a (transient or permanent) fault and the redundant execution is repeated. One example for a system implementing redundant execution is the SWIFT system [RCV+05].

Using reliability annotations, obviously, redundant execution is only required for sections of code that handle data annotated as reliable or unannotated data, if data without annotations are interpreted as reliable. Thus, the overhead in execution time caused by redundant execution only affects a fraction of the overall code base; consequently, the system will be able to fulfill real-time tasks with a reduced processing speed, which in turn will reduce the energy consumption of a system.

3 Static Analysis

The use of static analysis methods for the type annotation approach is recommended for two reasons: On the one hand, semantic analysis [ALSU07], including type synthesis and checking, is essential for valid programs. Any operation on an object with unreliable type will lead to a result that has to be considered unreliable as well. Consequently, a result can only be reliable if all operations and operands are reliable. Thus, semantic analysis has to check at compile time whether the data type annotations are used consistently according to this rule.

Additionally, certain operations are critical and require reliable execution, since the use of incorrect results may lead to irreversible consequences like a system crash. For example, objects with an unreliable data type must not be used in address calculations. These constraints can be verified with semantic analysis at compile time and mistakes by the application's developers can be prevented before they can have any effect.

On the other hand, considering all the above mentioned constraints, it may be difficult for the application developers to decide which data objects could be assigned an unreliable type and which data objects must not. Here, static data flow analysis could make suggestions which data objects may have an unreliable type. Of course, the creation of data annotations cannot be automated completely, since this would require application knowledge that cannot be extracted from the source code by analysis tools. However, the analysis

can provide valuable hints to application developers who finally assign annotations to a set of data objects from the candidates proposed by the analysis.

3.1 Semantic analysis

Semantic analysis is a common phase in most compilers, especially in compilers for the language C. In this phase, the compiler checks the semantic constraints given by the language specification. This includes type synthesis and type checks.

During type synthesis, the data type of expressions is calculated from the types of the subexpressions or data objects the expression is composed of. Afterwards, the compiler checks whether the types of the expressions are consistent with the requirements given by the language specification. Depending on the use of an expression, it may have to have a certain type, e.g., an array index shall be of integer type.

Since we propose the introduction of an *unreliability* annotation for data types, the type synthesis and the type checks performed by the semantic analysis have to be extended. In the remainder or this section, we describe the rules that specify the use of this new annotation.

3.1.1 Type synthesis

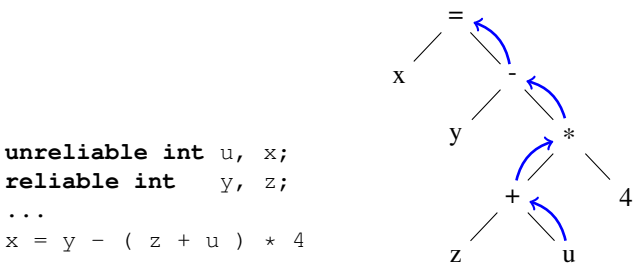
In general, we cannot decide whether a data object with unreliable type has its expected correct value or whether its value has already been changed by a transient fault. Thus, we always have to assume that a data object with unreliable type may have an incorrect value. When this data object is used in a computation, like shown in Figure 2(a), the effect will propagate to the result.

The propagation is illustrated in Figure 2. Here, the abstract syntax tree of the expression on the left hand side is shown. Since the data type of u is unreliable, the result of the addition has to be considered unreliable as well. Since the result of the addition is used for the multiplication, the result of the multiplication may be inexact as well. This continues until the root of the syntax tree is reached.

As a consequence, the type of an expression is unreliable, if at least one of its subexpressions[1] has unreliable type. Hence, whether the type of an expression is unreliable, can be calculated on the basis of the abstract syntax tree (AST) in a bottom-up approach. Every child node transmits an unreliable annotation to its parent in the AST. Finally, the result of the expression has to be considered as unreliable, if the root node has unreliable type.

There is only one exception from that transmit-rule. We allow the use of explicit C-like casts. Application developers can use casts to convert the unreliable data type of an object to a reliable one. Hence, such a cast will block the transmission of the unreliable annotation to the parent node in the AST. This enables the use of an unreliable data object, where a reliable data object is required by the language specification. The value of variable u in

[1] We also consider the use of a data object as a simple expression.



(a) Assignment (b) Abstract syntax tree

Figure 2: Propagation of the *unreliable* attribute

```
unreliable int u, z, values[16];
reliable int   r;
...
r = (reliable int) u;
if ( r < 0 ) || ( 16 <= r )
  // Value of r is outside the valid range.
  // Replace its value by a default value
  // and avoid expensive error correction.
  r = 7;

z = values[ r ];
```

Figure 3: Code example

Figure 3, for example, that may be affected by a transient fault is used after a range check has approved that its value is within a valid range. Despite the cast from the unreliable variable u, r is considered as a reliable variable by the analysis and can be used as an index whose possible values must match the array size. Since z is a variable of type `unreliable int`, the application developer may decide that an inexact array index may be tolerable here. The result of the indexing operation would even be considered as reliable by the analysis, if values was an array of `reliable int`, because the resulting type depends only on the base type of the array.

3.1.2 Type checking

The intention of semantic rules is not to restrict the possibilities of an application developer or to complicate the use of the programming language. Instead, they should exclude senseless constructs and reduce the risks of bugs.

According to this aim, we restrict the use of unreliable data objects to avoid the unintentional propagation of faults in an application by corruption of reliable data objects. Hence, the value of an unreliable data object shall not be assigned to a data object with reliable type. As shown in Figure 4, the assignment in line 4 can have fatal consequences. If u

```
unreliable int u, pos, tmp;
reliable int   r, a[10];
u = 10;
r = u;               // invalid assignment
pos = 0;
while ( pos < r )    // invalid condition
{
  tmp = r / u;       // invalid division
  a[ pos++ ] = tmp;  // invalid memory access
}
```

Figure 4: Negative example

is affected by a fault, its value may be higher than 10. Hence, the while-loop will iterate more than ten times and the content of a data object following array a in memory will be overwritten. Finally, the data objects corrupted by the out-of-control memory accesses can cause the application to crash, since, e.g., the value of a pointer has been changed and an invalid memory address is accessed.

Nevertheless, application developers can express their conscious will to treat a possible corrupted data object as reliable by using the cast operator as shown in Figure 3. The other way round, it is always possible to assign the value of a reliable data object to an unreliable data object without using a cast.

Also, the new semantic rules should inhibit the impact of uncorrected transient faults on critical parts of an application. As critical parts we consider expressions like address calculations, control-flow controlling conditions, and divisions.

Address calculations are critical, since an incorrect memory address will result in an arbitrary change of the program state. Such a fault can only be corrected by returning to a previously stored correct program state. Since the complete state has to be restored in this case, a lot of effort is required. This is contrary to our resource-conservation objectives for fault tolerant cyber-physical systems.

Transient faults may also affect the control flow of an application. Conditions that may alter the control flow, like loop conditions, carry the risk of incurring significant changes to the control flow. As a consequence, the program state may be corrupted, data may be processed incorrectly or loops may repeat endlessly.

Also, certain arithmetic calculations may be affected. For example, in divisions, the divisor must not equal zero. By not marking the divisor as unreliable, the risk of raising an arithmetic exception can be reduced. Thus, the code shown in Figure 4 will be rejected by the semantic analysis because of the invalid use of an unreliable data type in four spots.

3.2 Annotation hints

Annotating data types can be a considerable amount of work for large applications. Adding an annotation to the type at a single declaration may require annotations of data types at other declarations or may lead to a violation of the rules mentioned above. For an application developer, such sequences may be unpredictable. As a consequence, the application's developer may repeatedly add and remove annotations in a trial-and-error manner until he finds a valid annotation or he might even give up. Thus, it is desirable to use automated tools to support application developers by providing annotation hints.

First of all, an automated tool could create annotations for all the data objects that have to be reliable according to the rules described in section 3.1. Therefore, the tool would iterate over all expressions in the source code. Whenever an expression is used in a context that requires a data object with reliable type, such expression is marked as *must reliable*. As an example, all expressions used for address calculations will be marked this way.

Following, in a second step all data objects that are used in computations for such expressions are marked as *must reliable* as well, since the result of an expression can only be considered as reliable if all operations and operands are reliable. This step is repeated until a fixed point is reached, i.e., no additional expression or data object has been marked as *must reliable* in an interation.

Finally, the tool can generate explicit reliable annotations for those data objects that have been marked as *must reliable*. It is obvious that the application developers can focus on the annotation for the remaining data objects after this step.

Still, there may be data objects that have to be reliable even if the tool has not created an annotation for them, since their meaning for the application cannot be inferred from the source code alone. Certain data objects may have a major importance for the accuracy of the results, like the luminance level for a video image. If this factor is affected by a transient fault, all images may appear entirely as black or white. The application developers can refine the annotation by adding reliable annotations for data objects, where deviations from exact values cannot be tolerated.

If the application developers were confident that they had not forgotten to annotate a data object as reliable that must be corrected in case of a transient fault, unreliable data types could be used for all remaining data objects. However, the developers might proceed in a step-wise manner. First, they will annotate those data types with unreliable for which it is obvious that lower reliability requirements are sufficient. Applying data flow analysis, a tool can determine all expressions that are affected directly and indirectly by these changes. Then, the application developers can inspect which expressions can become unreliable and accept or discard the according changes.

The tool can also output suggestions to which objects the application developers might add the unreliable annotation next. Candidates are data objects that are used for the computation of the value of data objects with data having unreliable type. Of course, the tool will not suggest data objects already annotated as reliable.

We believe that such a work flow will reduce the work load of application developers and improve the utilization of our proposed annotations. Consequently, the efficiency of the approach is increased and the effort that has to be spent for error correction can be reduced.

4 Prototypical Implementation and Evaluation

In order to assess the viability of our approach, we implemented a first prototype of reliability-aware data types using a C preprocessor-based approach. We annotated some of the variables used for calculations in an inverse discrete cosine transformation (IDCT) (see Figure 5) as unreliable. All the other, unannotated variables were treated as reliable. Using preprocessor macros for reliable and unreliable annotations, we added a GCC-specific storage segment attribute to all variables marked as unreliable.[2] A modified linker script was then used to map the section containing unreliable data to a separate memory range.

```
#define unreliable(type, #args) type, args \
        __attribute__ ((section ("unreliabledata")));

unreliable(int x0, x1, x2, x3, x4, x5, x6, x7);

void njRowIDCT(int* blk) {
    ...
    x0 = (blk[0] << 11) + 128;
    x8 = W7 * (x4 + x5);
    x4 = x8 + (W1 - W7) * x4;
    x5 = x8 - (W1 + W7) * x5;
    x8 = W3 * (x6 + x7);
    x6 = x8 - (W3 - W5) * x6;
    x7 = x8 - (W3 + W5) * x7;
    x8 = x0 + x1;
    x0 -= x1;
    ...
}
```

Figure 5: Excerpt from the JPEG decoder

In order to simulate faults, we reused the fault injection approach employed for our H.264 decoder study [HESM10a]. A thread running in parallel to the JPEG decoder determined the memory start and end address of the *unreliabledata* section resp. the complete sections of initialized (*data*) as well as uninitialized data (*bss*) and randomly[3] inserted single-bit faults into that memory range. Injecting faults only in *unreliable data* simulates ECC-protection of all other memory regions.

[2] Due to the compiler modifications required, the attribute only works for global data.

[3] The injection target address and the bit affected by XOR were chosen using the Unix `rand()` function with a random seed. The target address offset was restricted to the respective memory range using a modulo operation.

Fault injection interval (μs)	10	20	40	60	80	100	200	400	600
Preliminary termination (number in 10,000 runs)									
unreliable	0	0	0	0	0	0	0	0	0
data+bss	1994	2334	2291	1819	1316	495	15	1	0
Incorrect results (number in 10,000 runs)									
unreliable	1154	1139	1013	719	537	395	365	242	205
data+bss	7928	7602	7625	7918	8050	8080	3811	2271	1308

Figure 6: Effects of simulated fault injection into unreliable data and data+bss

In order to evaluate the effectiveness of our approach, we compared the results to executions with errors injected all over the data+bss segment of the JPEG decoder. Since we intend to catch the most critical errors, we counted the number of program crashes that occured. The injection experiments were performed on a single core of an eight-core Xeon 2.53 GHz Linux system.

These first simulation-based results, shown in Figure 6, already look very encouraging. We ran each test 10,000 times on the same source JPEG image with fault injection in data+bss or in data marked as unreliable only. In order to easily observe errors, we varied the injection rate from one fault per ten microseconds up to one fault per millisecond. Since the complete runtime of the JPEG decoder is only about 12 milliseconds on our system, the results vary as expected.

We considered two parameters, the number of terminated decoding processes and the number of incorrectly decoded images after successful runs of the JPEG decoder. In all experiments, restricting the fault injection only to unreliable data avoided all preliminary terminations of the JPEG decoder. In contrast, injecting faults all over data+bss resulted in termination rates of 5%–20%. Comparing the results, restricting the fault injection to unreliable data reduced the number of incorrectly decoded images for the same injection rate by up to a factor of 15.

5 Related Work

Semantic type qualifiers were first proposed by Chin et al. [CMM05]. In this paper, the authors describe the use of user-defined type refinements to augment existing types in order to statically ensure additional invariants. Some examples for these type refinements include nonnull, nonzero, untainted, and unique refinements. In addition, the authors developed an extensible typechecker and a soundness checker to ensure the respective invariants. The *tainted* annotation discussed in the paper is similar to our reliability annotation – declaring a variable as untainted implies that a data flow resulting in values for that variable may never encompass any values marked as tainted. Essentially, the taintedness could also mark variables in unprotected memory, so this approach would also be useful for implementing fault tolerance.

Classification of data to optimize systems according to given objectives is a relatively new research topic. A recent publication by Sampson et al. [SDF+11] describes the use of approximate data types in applications that are able to tolerate certain inaccuracies in order to conserve energy. Similar to our approach, they propose the use of type qualifiers to identify the precision requirements of a data element. However, the authors concentrate on Java as implementation language, which restricts using the approach in small-scale cyber-physical systems. While the authors achieved impressive results, potential savings of 10%–50%, they do not consider implementing static analysis methods to verify the correctness of the annotations. In addition, only probabilistic approaches (like we intend to handle in a future project, see section 6) are considered; transient faults seem to be outside the focus of the authors' work.

Perry et al. [PMR+07] propose the use of typed assembly language to handle transient faults. Their approach explicitly marks reliability-critical data in an assembly language program and adds proactive replication when code handles that data. While this interesting approach provides an effective method to handle extremely critical transient faults, the use of assembly language seems exotic for most of today's cyber-physical systems. However, a low-level approach using typed assembly language might well complement our approach if a suitable compiler backend that considers the reliability annotations would be developed.

Chalin describes the use of nonnull annotations in Java 5 [Cha06]. Intended as a tool for improving software reliability, nonnull annotations are used in concert with static analysis methods to ensure that null values are not used. In addition, that paper proposes an extension of the Java Modeling Language to support a default non-null semantics. Like several earlier approaches, however, these annotations are used to ensure semantic program correctness but are not prepared to handle asynchronous unexpected errors.

6 Conclusions and Future Work

Our experiments show that using reliability annotations can reduce the error correction overhead in a system while reducing the amount of redundancy that has to be introduced. In this paper, we evaluated the effectiveness of our approach and were able to avoid all crashes when restricting fault injection to variables manually identified as non reliability-critical and annotated accordingly. However, we are aware that our concept is only the first step in providing a flexible error handling approach based on reliability annotations.

The encouraging results of our first prototype will now have to be validated by implementing the annotations in the context of a real compiler front end; in addition, the static analysis methods described in section 3 have to be implemented in order to automatically provide annotation hints. We will use the ICD-C framework [ICD] to perform code transformations and static analyses. Further steps required are the extension of our mapping approach to separate memories to local variables residing on the stack, which requires extensive back end modifications. Future work will also include the application of reliability annotations to multicore mapping, redundant code execution and possibly further redundancy approaches.

We also intend to evaluate the applicability of our annotation approach to other areas of reliability. One very promising technology is probabilistic computing. Here, specific parts of system components, like the low-order bits of an adder [LLCB10] or a multiplier [GMM+11], are supplied with reduced supply voltages in order to conserve energy. This results in the generation of incorrect results from these components, since signal propagation speed is reduced. In the case of an adder, certain long carry chains are not propagated in time, leading to a loss of precision.

Here, we intend to apply our classification and data annotation in a cooperation with Nanyang Technological University, Singapore, which develops such probabilistic CMOS (PCMOS) devices and error models. In contrast with the transient faults discussed in this paper, it is well known which operations of a system will be affected by this probabilistic behavior. As with transient faults, however, it is not generally predictable at which point in time (i.e., which point in the execution of an application) an error might show up. The implications for our reliability annotations will be interesting to explore.

References

[ALSU07] Alfred V. Aho, Monika S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2nd edition, 2007.

[Cha06] Patrice Chalin. Towards Support for Non-null Types and Non-null-by-default in Java. In *Proceedings of the 8th Workshop on Formal Techniques for Java-like Programs (FTfJP'06)*, 2006.

[CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 85–95, New York, NY, USA, 2005. ACM.

[GMM+11] Arjun Gupta, Vincent J. Mooney, Satyam Mandavalli, Keck-Voon Ling, Arindam Basu, Henry Johan, and Budianto Tandianus. Low Power Probabilistic Floating Point Multiplier Design. In *Proceedings of ISVLSI 2011*, Chennai, India, 2011.

[HESM10a] Andreas Heinig, Michael Engel, Florian Schmoll, and Peter Marwedel. Improving Transient Memory Fault Resilience of an H.264 Decoder. In *Proceedings of the Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia 2010)*, Scottsdale, AZ, USA, October 2010. IEEE Computer Society Press.

[HESM10b] Andreas Heinig, Michael Engel, Florian Schmoll, and Peter Marwedel. Using Application Knowledge to Improve Embedded Systems Dependability. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep 2010)*, Vancouver, Canada, October 2010. USENIX Association.

[ICD] ICD e.V. The ICD-C Compiler Framework, `http://www.icd.de/es/`.

[ITR] ITRS. Intl. Technology Roadmap for Semiconductors, `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf`.

[LLCB10] Mark S. K. Lau, Keck-Voon Ling, Yun-Chung Chu, and Arun Bhanu. A General Mathematical Model of Probabilistic Ripple-carry Adders. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1100–1105, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[PMR+07] Frances Perry, Lester Mackey, George A. Reis, Jay Ligatti, David I. August, and David Walker. Fault-tolerant Typed Assembly Language. *SIGPLAN Notices*, 42:42–53, June 2007.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code generation and Optimization*, CGO '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[SDF+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, and Luis Ceze Dan Grossma. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, San Jose, California, USA, 2011. ACM.

[WCS09] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Mixed-mode multicore reliability. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 169–180, New York, NY, USA, 2009. ACM.

[YPPJ06] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended Protection against Stack Smashing Attacks without Performance Loss. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 429–438, Washington, DC, USA, 2006. IEEE Computer Society.