technische universität
dortmund

department of
computer science

TECHNICAL REPORTS IN COMPUTER SCIENCE

Technische Universität Dortmund

Passing error handling information from a compiler to runtime components

Florian Schmoll, Andreas Heinig, Peter Marwedel, Michael Engel

Computer Science 12 – Design Automation of Embedded Systems Group

Number: 844
May 2014

Technische Universität Dortmund — Fakultät für Informatik
Otto-Hahn-Str. 14, 44227 Dortmund

http://ls12-www.cs.tu-dortmund.de

Florian Schmoll, Andreas Heinig, Peter Marwedel, Michael Engel: *Passing error handling information from a compiler to runtime components,* Technical Report, Department of Computer Science, Dortmund University of Technology. © May 2014

ABSTRACT

For the handling of faults in embedded systems, software implemented fault tolerance seems to be more appropriate than hardware based approaches. Using software based techniques, runtime conditions only known to software can be considered. Also, the error handling can be application specific and there are more alternatives for decisions during error handling resulting in a more flexible approach. By adapting the error handling to the requirements of the software resources can be saved.

A recent publication [1] showed that a compiler that evaluates source-code annotations and applies static analyses can determine which errors require error handling, and which errors can be ignored safely. This error handling information can improve the efficiency of software implemented error handling. Additional information about how erroneous data can be handled is also provided by source code annotations. However, error handling information is needed at runtime, when error handling actually takes place. Unfortunately, the computation is too complex for embedded systems so that it cannot be computed on demand. Hence, the relevant information has to be precomputed at compile time, but must be retrievable at runtime.

In this report we present how compiler generated information about the error handling of data objects can be made available to runtime components that apply error correction.

ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500) under grant no. MA-943/10.
http://spp1500.itec.kit.edu

CONTENTS

1 INTRODUCTION

Embedded systems often have to be cost-efficient [2]. By continuous miniaturization of the structure sizes in information processing devices, more cost efficient systems can be produced. Other properties of the system, like energy-efficiency, can be improved in this way as well. In contrast, the inherent resiliency of the hardware against memory faults is reduced. Therefore, increased rates of faults are expected [3] that require countermeasures.

A memory fault is a violation of the premise that a read of a memory cell always returns exactly the value that has previously written to it[1]. Typically, permanent and transient faults are distinguished. Permanent faults are hardware defects that result from an incorrect structure of the hardware due to production faults, aging, or wear out. Hence, permanent faults mainly occur in new or old systems. Their distribution over time can often be characterized by a bathtub curve. Examples for permanent faults are stuck-at-zero or stuck-at-one faults.

In contrast, transient faults, also referred to as Single Event Upsets, do not affect the operability of components. They are reversible, accidental changes of the states of components that are triggered by disturbances that impact on the system from the exterior, like radiation, or emanate from the system itself, like heat. Hence, transient faults can affect a system during its whole lifetime and cannot be ruled out by post-production tests. An example for a transient fault is a spontaneous bit-flip in memory. In the following we will consider only this kind of fault.

A memory fault that can influence the operations of a system, because it affects data that may be used in the future, is also referred to as error. The effects of an error can be diverse depending on the semantics of the data that has been affected. If the error changed the value of a memory address, it may lead to an invalid memory access that results in a termination of the application software. Also, errors can change the control flow of an application, thus changing the stream of operations executed by the system and its timing behavior. Additionally, the error can affect the precision of the outputs computed by the application to different extents.

The occurrence of transient faults can hardly be avoided. E. g., to avoid faults that are caused by radiation, an expensive shielding would be required. Hence, systems are not protected against errors, but against their effects. Therefore, error detection and correction techniques have to be applied. Error detection techniques have to determine erroneous states, before they are processed by the system and can evolve their unwanted effects. To enable checks for errors, along with the actual memory value, redundant information, like a checksum or a copy of the value, has to be stored. Error correction techniques permanently avert the danger of the error effects by eliminating the error. Errors resulting from transient faults only manifest as erroneous states. Hence, they can be eliminated by replacing the erroneous states. Obviously, a system with Error

1 The write need not originate from the application software, but may be performed by I/O-devices.

Detection and Correction (EDAC) requires more hardware resources than a system without. Since additional hardware increases the production costs of the system, EDAC techniques for embedded systems are not only assessed based on how good they can eliminate the effects of errors, but also based on their resource consumption.

For the detection and correction of errors both hardware and software approaches are feasible. Hardware approaches have the advantage that they are transparent to the application software. Hence, the software need not fulfill special requirements nor has to be modified. This is favorable for error detection, where checksums have to be computed and compared at every memory (read) access. In software, several instructions are required to implement these operations resulting in undesired processor load and an increase of code size. However, hardware approaches are unaware of the resource usage of the application and the semantics of data. Consequently, they have to provide EDAC for the whole memory, leading to a permanent overhead. Nevertheless, the correction capabilities are often limited to a single or small number of bit-flips per memory word. Furthermore, special hardware components are required increasing the costs of the system.

Software approaches also require additional hardware resources, e. g., for the storage of redundant information that is needed for the detection of errors and the recovery of correct states, or the compensation of the additional processor load. However, no special hardware is needed and more powerful general purpose hardware can be used. Hence, there is no strict separation between components for EDAC and for the actual tasks of the system. Memory and processor time reserved for error correction can be used otherwise reasonably in case no error shows up. Additionally, software approaches can be more flexible than hardware approaches. There are approaches that can correct multi-bit errors and that restrict EDAC to those parts of an application that are most sensitive to errors. Hence, for the implementation of a resource efficient EDAC, a combination of error detection in hardware and a correction of errors by a software approach seems promising.

For extra efficient error correction the FEHLER approach has been developed. It only considers the correction of data, since it assumes that errors affecting instructions can be corrected by reloading the corrupted instructions from the software image that is stored in a fault-free ROM. The software approach allows for a flexible correction of errors, i. e., the appropriate correction method for an error is selected at runtime, when the error has been detected. In this way, the possible effects of an error and runtime conditions like the available resources can be taken into account and the effort for the correction of the error can be adapted to these factors. This is a feature that is neither achieved by hardware approaches nor by existing software approaches. To determine the possible effects of an error, the source code of the applications executed on the system has to be analyzed. Also, information about how errors can be corrected and the properties of correction methods are determined at compile-time. This error handling information has to be passed to the components that select and execute the error correction methods at runtime.

In this report, we show how this passing of information can be implemented efficiently. We start with a more detailed presentation of the FEHLER approach in the next chapter. There, the library for runtime error classification *librecon* is

introduced that is the central component that implements the passing of error handling information. Also, the components that interact with the library are presented. The way in which the runtime components interact with *librecon* is defined by uses cases. They will be described in chapter 3. The implementation of the runtime error classification including the encoding of the error handling information is topic of chapter 4. The creation of the *librecon* by an automated workflow is presented in chapter 5. In chapter 6, the similarities and differences between the problem of passing error handling information and the problem of passing debug information are pointed out. Finally, a conclusion and an outlook are given in chapter 7.

2 FEHLER APPROACH

The FEHLER approach is motivated by the observation [4] that errors have different effects. Some errors can cause malfunctions that influence the availability of the services provided by a system, e. g., lead to a termination of an application. Some errors can reduce the quality of service of the system, e. g., they lead to jitter in the output stream. Other errors have only negligible effects, e. g., they lead to minor deviations in the computed outputs. The complete elimination of the effects of an error is typically effortful. Multiple copies of data values or correction codes have to be maintained, so that a fault-free version for the correction exists, or data values have to be backed up regularly so that a previous fault-free state of the system can be restored. These options are both memory- and time-consuming.

The idea of the FEHLER approach is to have several correction options available. Some errors may be correctable more efficiently with special correction methods than with general methods. Correction methods may not completely eliminate the effects of errors, but mitigate them and at least ensure that the remaining consequences are not hazardous. Additionally, the effects of an error may be within a tolerable extent, so that the presence of the error can be accepted and a correction can be completely ignored. Such correction methods can help to save a scarce resource budget, although compared to a complete elimination of an error, a degradation of the quality of service is possible.

The several correction options allow for a flexible handling of errors and enable a trade-off between correction quality and resource consumption. The way an error is corrected can take its presumed effects and the amount of available resources into account. Consequently, errors affecting the same memory location at different points in time can trigger different correction methods. E. g., whether the effects of an error are eliminated completely with a complex correction method or are only sufficiently mitigated with a skeleton correction, can depend on the available processor time for the correction that remains in the schedule of the system.

An implementation of the FEHLER approach consists of three major components: A compiler, runtime components, and a runtime error classification. The compiler determines the feasible correction options for an error using a data centric approach. The possible effects of an error are computed starting from the data objects the error affects. A data object is any piece of memory that is represented by a symbol in the source code. Hence, a data object can be a simple scalar variable, an array, or struct, but also a variable that is defined as part of a struct. For these objects different correction options exist, depending on their use and semantics. Thus, an error affecting a struct can be corrected either by one of the options for the whole struct, or by one of the options for the nested object to which the error is limited. Additionally, whenever an erroneous memory range has to be corrected, a complete data object that is allocated to that range has to be considered. A correction that is unaware of the objects that cover the range can lead to a partial modification of the objects and itself can

result in inconsistent states. Therefore, error correction methods are related to objects.

The use of data objects is specified in the source code of an application. Hence, the computation of possible error effects resulting from the use of erroneous data objects requires source code analysis [1], which necessitates the use of the compiler. However, the semantics of data objects and correction methods can hardly be understood by a tool. Therefore, we assume that this information, like applicable correction methods for data objects, are specified in source code annotations. The compiler evaluates these annotations again. Based on an initial set of annotations the compiler can add annotations by itself using automated reasoning, so that the effort for inserting annotation into the source code can be reduced. All the information computed and collected by the compiler makes up the error handling information. It includes the information about the correction options, but also features of the available correction methods, like their capability to eliminate error effects and their resource usage.

The runtime components implement the error correction. To enable a flexible error correction a detection of an error does not automatically trigger its correction. Instead, the runtime components are signaled that begin with the identification of the affected data object. The available correction options can be retrieved from the error handling information. The runtime components are aware of the resource scheduling in the system, so that they can assess the options regarding the current runtime conditions in the system, like the execution time available for correction and the count of errors whose effects have not been eliminated completely. Finally, the runtime components select and execute the appropriate error handling action. Precautions ensure that the runtime components continue operable, even in case they are affected by faults.

Although the error handling information is actually needed for the selection of the error correction at runtime, when errors show up, it cannot be determined by the runtime components. Because of the complexity of the computations, for all data objects that can potentially be affected by an error, the error handling information has to be compiled in advance of its use in the error correction. Otherwise the timing behavior of the system would be seriously disturbed. Also, for the computations, the source code of the software is needed, which is typically not accessible at runtime. Hence, the error handling information that is computed at compile-time must be retrievable by the runtime components. This problem is solved by a runtime library, that we denote *librecon*, where RECON is the abbreviation for Runtime Error Classification. It is created by the compiler and stores the error handling information. The library also contains interface functions that enable the runtime components to access the required information without the need of knowing how the information has been encoded.

3 USE CASES

The runtime error classification library, *librecon*, has to support three use cases. In the first use case, the runtime components that carry out the error correction employ the library to determine the data objects that may be affected by a detected error. Additionally, the runtime components request error handling information that is required to directly respond to the error. In this way, an uncontrolled propagation of the error to other data objects can be prevented that may result in a corruption of the system otherwise. In the second use case, the runtime components retrieve information about the available correction methods for the individual data objects. The retrieval of information about the error correction methods is subject of the third use case. In the following, these three use cases will be described in more detail.

3.1 DATA OBJECT IDENTIFICATION

A request for object identification by *librecon* is preceded by the detection of an error. The error detection leads to a notification of the runtime components which, in turn, locate the position of the error in memory. For two reasons, the runtime components may be unable to determine the exact position of the error. On the one hand, the error detection itself may be unable to distinguish single bytes. To reduce the number of redundant bits that are required for detection, it may operate on larger blocks of memories, to that we refer as error detection sections. The error detection can determine the existence of an error within the section, but not its position. Consequently, the extent of these sections directly impacts the precision of the error localization. On the other hand, operations that are not directly executed by software, like cache line fetches or DMA, can consist of a sequence of accesses to several error detection sections. An error detection unit may signal only that an error was detected during an operation, but not report the position of the affected section(s). In this case, the runtime components have to deduce the position of an error from the memory range that should have been read by the operation that failed. Thus, it remains unclear in which accessed section an error had been actually detected. Hence, the result of the localization can be a range of memory addresses.

For an object identification request, this range is then passed to *librecon*. Although only a single bit within the range might actually be erroneous, in the absence of more precise information, *librecon* has to assume that the whole range is potentially affected by the fault. Hence, the task during object identification is now to determine all data objects that are allocated either completely or partially to the memory range.

Since at compile time only the allocation of global variables is known, *librecon* can identify only this kind of data objects. On the contrary, base addresses and memory layout of global variables are usually unknown to runtime components. Type information is already considered by the assembler code generated by the compiler and a linker already inserts the address values for the accesses

to the variables. Consequently, runtime components are unable to map memory addresses to global objects. Thus, it is actually necessary that *librecon* supports the identification of global variables.

In contrast, the position of dynamically allocated data in memory is set at runtime and hence, unknown at compile time. Therefore, heap objects have to be identified by the runtime components that do the dynamic memory management. Nevertheless, *librecon* can provide information about the memory layout of these objects, but this implies that a type is assigned to heap objects and any use of the object in the source code complies with this type. In particular, the type of the allocated object has to be unambiguously determinable.

Even though the structure of the stack frame for each function is known at compile time, identification of stack data is currently not supported by *librecon*. The final memory addresses of stack data at a specific point in time depend on the initial stack pointer of the application and the functions that have been called, but have not returned so far. This history of function calls can depend on input data and hence, is hardly predictable at compile time.

We assume that global variables, heap data, or stack data are allocated to separate memory ranges and that ranges with different kinds of data are not part of the same error detection section. Thus, for each memory range it is clear which component performs the object identification.

The result of the data object identification is a list of datasets about the objects that are allocated to the error detection section for that an error has been detected. Each data set contains the start address and the size of the object, a list of task names, and a unique key. The start address and the size of the object are required for error correction. They allow correction methods that correct individual objects the determination of the range that has to be corrected. The list of task names specifies which tasks might use the data object. By suspending only these tasks, the runtime components can inhibit accesses to the erroneous data, while the execution of unaffected tasks can be continued. In this way, errors cannot influence the computation of other data objects and the propagation of errors can be prevented effectively. Finally, each object has a unique key that allows for a faster identification of the object than by its memory address. It enables an efficient lookup of error handling information for the object, as it is the subject of the second use case.

3.2 RETRIEVAL OF ERROR HANDLING INFORMATION

After the objects affected by an error have been identified, information about error handling options for these objects can be queried. This takes place in the separate, second use case that is described in the following. The provided information should support the runtime components in selecting the correction methods that are most appropriate for the trade-off between quality of the correction and the required effort. Therefore, *librecon* contains information about the data objects themselves as well as the available correction methods.

librecon distinguishes between application and system objects. For application objects, their use in the source code can be analyzed. Also, data annotations in the source code can be evaluated. In contrast, system objects are also used beyond the scope of the application. These objects are often part of system components for which no source code is available, or whose source code can

hardly be analyzed statically. Hence, a thorough analysis of these objects is not possible and pessimistic assumptions are made. Consequently, more precise information is available for application objects.

For each data object, its *impact* on the system can be queried. It is an estimation of the degradation of the quality of service provided by the system if the object is affected by an error and the error is not corrected. It allows for a prioritization of errors during error handling if insufficient resources for the correction of all errors are available. Finally, for each data object the set of applicable error correction methods can be retrieved. This is the set from which the runtime components can choose a method to correct the erroneous data object.

3.3 RETRIEVAL OF ERROR CORRECTION METHOD PROPERTIES

For error correction methods, the runtime components require information to assess their correction quality as well as the effort needed to execute the correction. Therefore, *librecon* provides data about correction methods' properties, like their execution time and their impact on the quality of service. This information can be queried in the third use case that is presented in this section.

All correction methods that may be applied within the flexible error correction approach are marked as correction methods in the source code and annotated with information that can be queried in this use case. In doing so, all error correction methods are registered at compile time. In *librecon* the correction methods are enumerated. By passing the index of the correction method to *librecon*, the method can be identified and its properties are returned.

As already mentioned, the worst-case execution time (WCET) of a correction method is one of the properties provided by *librecon*. The correction quality of a method is described by its quality of service. Since correction methods do not necessarily have to restore the state of the data object that it had before it had been affected by the error, the methods can reduce the degradation of the quality of service that an uncorrected error may cause to a different extent. However, the actual impact of the correction on the quality of service is hardly predictable. Hence, the quality of service of a method has the purpose to enable a ranking of the correction methods by their capability to mitigate the effects of an error. Currently, the quality of service can be specified by one of the four levels: minimum, simple, good, and perfect.

Another property provided for correction methods is the fault factor. It is an estimator for the probability that the correction itself is affected by an error. The actual probability depends on the correction method, but also on properties of the system, so that it cannot be exactly specified. For example, the fault rate experienced at runtime has an influence on the probability, but can vary over time. Hence, the fault factor only considers properties of the correction method. This includes the execution time of the correction method, since a longer execution time increases the probability for an error, but also the resource usage of the correction method. Here, we assume that a correction method is more likely to be affected by an error the more frequently it accesses different resources like memory locations. As for the quality of service, the purpose of the fault factor is to enable a weighting of different correction methods.

Finally, for each correction method *librecon* provides the set of data objects that are used during its execution. This enables the modeling of dependencies

between the correction of errors. They should ensure that all data objects required by a correction method are error-free or have been corrected before the method is executed. At compile-time it is checked that for each global data object at least one correction method exists that will not result in a circular dependency between correction methods. This restriction to global variables results from the fact that besides the data object to be corrected, a correction method can only access global data.

4 IMPLEMENTATION

The implementation of the runtime error classification as library *librecon* has to take some requirements into account that result from its use in an embedded real-time system. Also, the potential occurrence of errors in memory has to be considered. The relevant implications for the implementation are described in Section 4.1. As pointed out in the previous Chapter, any request to *librecon* demands the successful identification of the data objects that have been affected by an error. In Section 4.2, the concept for the representation of data objects is presented. Finally, the data structures and algorithms used to efficiently identify the data objects and to store and provide the error handling information as defined by the use cases are described in Section 4.3.

4.1 REQUIREMENTS

One of the constraints of embedded systems that is relevant for the implementation of *librecon* is the restricted amount of memory available in these systems. Hence, the implementation of *librecon* has to be memory-efficient. It is very likely that pieces of error handling information for several data objects are the same, since they have the same memory layout, the same correction methods, or are used by the same tasks. To reduce the memory space required by *librecon*, multiple storage of identical information should be avoided.

Since *librecon* is used in real-time systems, its response time is also an important property. It directly correlates with the execution time of the functions provided by *librecon* to retrieve the error handling information. Because error handling includes requests to *librecon*, the time needed by *librecon* to determine the results has a direct impact on the execution time for the whole error handling. Consequently, the faster *librecon* can process the requests, the more time remains for the correction of errors, and the likelihood that deadlines can be kept increases. Thus, the implementation of *librecon* has to be time-efficient.

librecon is part of a system that can be affected by faults. Nevertheless, a fault must neither break the service provided by *librecon* nor make it unavailable, since it belongs to the error handling system. In this case, the system would not be able to recover from the fault. Thus, for faults affecting *librecon* there must be a simple way to completely eliminate their effects. Also, the future behavior of *librecon* must not be affected by faults. However, the correction of an error should not require the execution of a correction method. Otherwise, *librecon* would not be able to process requests until the correction method has finished. Deadlocks can occur if the determination of the appropriate correction for *librecon* results in a request to *librecon* that cannot be processed because of the error. Instead, if the runtime components detect an error during the execution of a function of *librecon*, they should be able to eliminate this error by a simple reexecution of the function. This requirement can be ensured by a implementation of *librecon* that is stateless, meaning *librecon* has only one immutable global state. However, this state has to be protected against errors. Since it does

not change, all the data that represents the state can be mapped to read-only memory, so that accidental write accesses to the data storage of *librecon* can be excluded. Also, we assume that read-only memory is not affected by faults. Nevertheless, if errors occur in the memory itself, the state of *librecon*, like any read-only data in the system, can be restored anytime by simply reloading the data from the system image. Strictly speaking this restore of the state of *librecon* is a correction method, against which we argued before because of possible deadlocks. However, the risk of a deadlock can be ruled out if *librecon* is queried only for the correction of errors that do not affect read-only data.

Finally, the implementation has to cover all the uses cases that have been described in the previous chapter. Thereby, *librecon* should be able to provide error handling information for each global data object of the applications. Specifically, composite data objects like structs and arrays and their components should be distinguished by *librecon*, so that separate information can be provided for the composite and the nested data objects. Depending on whether the complete composite object is affected by a fault or only a part of it, and depending on for which objects error correction methods have been specified, *librecon* should be able to consider different error handling scenarios.

4.2 DATA OBJECT REPRESENTATION

Decisive for the efficiency of *librecon* is the representation of the data objects. As described in Section 3.2, error handling information is only provided for data objects of the application. Each application object should be represented by a unique key $k \in \mathbb{N} \setminus \{0\}$ that allows for the distinction of data objects and the efficient retrieval of the error handling information. Since no error handling information is provided for non-application objects, they all can use the same key: zero. The base addresses of the data objects in memory are inappropriate keys. They are known only for global variables and they are not unique. For example, the base address of a composite object like a struct and the address of its first nested object are the same. On the one hand, the choice of the object representation directly influences the object identification (see use case one), that has to map memory addresses to the object keys.

On the other hand, for the retrieval of the error handling information for the individual objects, as described in use case two, the keys have to be mapped to the corresponding datasets. If the keys form a range of consecutive numbers with only few gaps, this mapping can be implemented with an array that is known to be memory efficient and to allow for fast accesses. The fewer gaps are in the range formed by the keys, the less unused elements will be contained in the array. In the following we will present, how the data objects can be represented by the keys from the range $[1 \dots n]$, where n is the number of data objects to be distinguished. In the next section we will then show that the assignment of keys to objects perfectly matches the object identification, i. e., the mapping of memory addresses to keys.

We denote objects that are not nested in another object as top-level objects. Each of the top-level objects requires a unique key by which the global variables of the applications can be distinguished. The assignment of the keys to the top-level objects becomes obvious, when the key assignment of nested objects has been explained. Nested objects represent a part of a composite object

Listing 1: Declaration of source objects

```
struct S1
{
  int sub_a;
  int sub_b;
};

struct S2
{
  struct S1 x;
  int       y;
  struct S1 z[ 3 ];
};

struct S2 top_a;
struct S1 top_b;
```

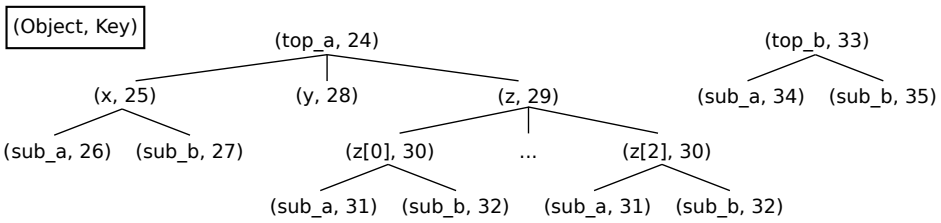

Figure 1: Example assignment of keys to objects represented by pairs of object name and key value

depending on its memory layout. Arrays are composite objects that represent a sequence of homogeneous objects. Therefore, we assume that error handling information for these objects is homogeneous as well. Hence, the same key will be assigned to all these objects. This reflects that we consider the complete array and the elements within the array as two separate objects. In contrast, struct-like composite objects contain heterogeneous objects. Here, a different key should be assigned to each nested object. This is analogous to the declaration of data types in C. Like for the objects shown in Figure 1 that result from definitions shown in Listing 1, to each object with a different symbol another key has to be assigned. Also objects in two different instances of objects with the same composite type, the objects represented by the same symbol, like the nested object `sub_a`, require different keys. Hence, the object `sub_a` in `top_a` has another key than object `sub_a` in `top_b`.

To ease the reuse of the memory layout information about composite objects, the assignment of keys to the nested objects should always follow the same schema. We decided to assign consecutive keys to the objects that are part of the same composite object in pre-order (compare with Figure 1). This assignment avoids a fragmentation of the key range. Moreover, with the following condition it can be easily checked, whether an object is contained within

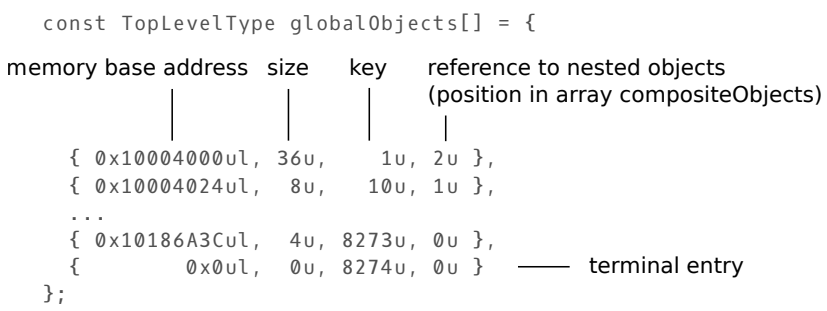

Figure 2: Encoding of memory address to object key mapping for top-level objects representing global variables

another object if for this object the total number of nested objects is known: a nested in $b \iff key(b) < key(a) \leqslant key(b) + \#nested\_objects(b)$.

For the assignment of keys to the top-level objects, the objects are sorted by their base memory address. Subsequently, the keys are assigned to the objects in this order. If an object is a composite object, first keys to the object itself and its nested objects are assigned like described above, before the next top-level is considered. Hence, if there are two top-level objects a and b and there is no top-level object that is allocated between the objects, the following equation holds: $key(a) + \#nested\_objects(a) + 1 = key(b)$.

4.3 DATA STRUCTURES AND ALGORITHMS

The mapping of memory addresses to the keys of top-level objects is simple. It can be represented by storing for each object a dataset consisting of its absolute memory base address, its size, and its key. As for the assignment of keys, the objects are sorted by their base address. Their attributes are stored in this order in an array. This is illustrated in Figure 2. For a top-level object the count of nested objects can be determined by looking up the key of the next application element in the array and computing the difference. A terminal element in the array enables this way for the last top-level object. Objects within a memory range that has been affected by an error can be efficiently found using binary search. To stop the search early in the case that the affected memory range is within a section that no application objects are allocated to, *librecon* stores the start and the end of each section that contains application data objects and the array indices of the datasets of the first and last object in this section. The corresponding data structure is shown in Figure 3. Before the search in the array with the dataset for the objects is started, the section that contains the affected memory range is determined. Only if the search is successful, the object identification is continued. Since for each section the range of the datasets of objects that belong to the section are stored, the binary search can be limited to this range.

A more sophisticated approach than for the encoding of top-level objects is needed for composite objects. Admittedly, for each instance of a composite object a separate mapping can be used which is similar to a decomposition of the composite objects. However, this approach becomes impractical if the

```
const SectionType sections[] = {

     start address     end address   first index  last index
           |                 |             |           |
  { 0x10004000ul, 0x10006D23ul,      0u,    138u },
  { 0x1000D000ul, 0x1000D81Cul,    139u,    182u },
  ...
  { 0x10100000ul, 0x10186A4Ful, 14813u, 16829u }
};
```

Figure 3: Encoding of memory sections that are relevant to *librecon*

composite object is a large array. Also, a decomposition does not support the distinction between composite and nested objects. Anyhow, to save memory it will be beneficial if only one mapping for all objects with the same memory layout or type can be used.

To enable the reuse of the memory layout information for composite objects the position of the nested components must not be encoded by their memory address, since it will be different for components in two different composite objects. Instead, for each nested object the relative offset to the start of the directly enclosing object is stored. This implies that composite objects form a hierarchy analogous to the nesting relation of their types, as shown in Figure 1. A search for the identification of the nested object browses this hierarchy starting at the top-level object. Only the address of this object is an absolute memory address which is needed to compute the address of nested objects based on the sum of their relative offsets. Besides, the top-level object need not be a global variable. The approach for the encoding of the memory layout information can also be used for objects that are allocated to the heap or stack if the top-level object is already identified and its attributes are passed to *librecon*.

For the same reason as for the memory addresses, the keys of the nested objects cannot be stored explicitly. Depending on the object, to that the nested object belongs, the key for the object varies. However, since the way in that keys are assigned to the nested objects is the same for any composite object, the relative distance of keys to the key of the directly enclosing object is also the same. Hence, as for the memory addresses, the key of a nested object can be computed by adding the relative key offset of the object to the key of the composite object.

The hierarchy of objects has to be encoded as well. All objects that are contained in the same composite object form a group. Their datasets can be stored consecutively in an array as shown in Figure 4. The hierarchy of objects can then be represented by a reference in the dataset of the composite object to the position of the first element of the group in this array. The memory layout is only relevant for application objects. For all other objects, the reference value is set to zero. This implies that the first element in the array is just a placeholder and does not contain information about a nested object.

Objects that represent arrays also have a reference to the group representing its base type. It can be distinguished from a struct with only one element by a simple size comparison. If the size of the composite object is a multiple of the size of the group representing the base type, it is an array. In doing so,

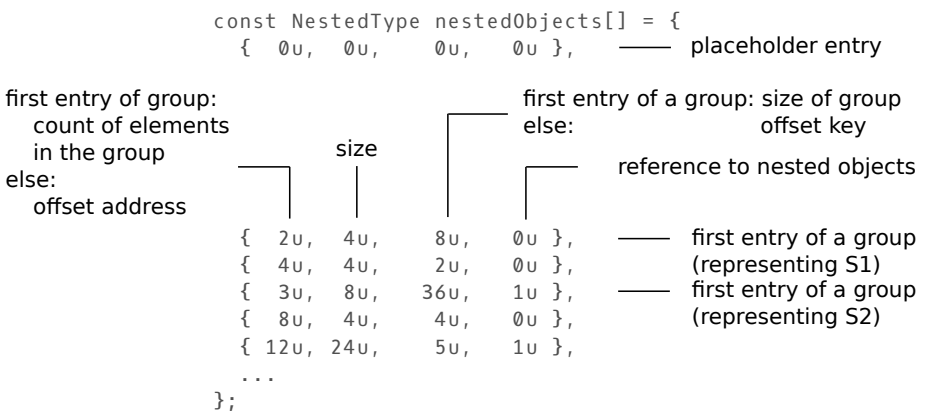

Figure 4: Encoding of the memory layout of composite objects

single objects and an array of length one cannot be distinguished. Anyhow, a distinction is not required, since it does not make any different for the identification of either the nested objects or the object itself. However, arrays that differ only in their size, but not in their base type, can share the mapping to nested components.

Error handling information for the application objects is assumed to be stored in an array of datasets. In case that the information contains elements of variable size, like the list of tasks that use an object or the list of applicable error correction methods, we assume that the dataset is designed to include the largest element size or these elements are outsourced. However, the size of the elements will not change at runtime. Also, the type of the elements that can have different sizes are known at compile time. Hence, they can be stored in an array as well, where there is an array for each type of variable-sized elements. E. g., lists can be stored one after the other in an array, where each lists starts with the count of its elements followed by the sequence of elements. The list can be simply referenced by the array index of the list head.

If the same error handling information is provided for more than one object, it should be encoded only once and shared between the objects. To enable the reuse of error handling information, a mapping of object keys to the array index of the error handling information is required. Adapting the keys instead, so that they are identical with the position of the information dataset in the array, will not work for our approach in general. Since there is a fixed relation between the keys within composite objects, their assignment must follow the same schema for each instance of a composite object and cannot be changed. Nevertheless, the mapping can be encoded efficiently. Since the domain of the mapping is the gap-less sequence of object keys, here an array can be used again.

5 WORKFLOW

The error handling information provided by the runtime error classification *librecon* is application specific. It refers to the data objects of a particular application. Hence, for each differing set of applications another version of *librecon* has to be created. Since we consider embedded systems, the set of applications will not change at runtime. Thus, if the software of the embedded system has to be updated or replaced, a complete new software image can be created by the same tool flow that created the initial software image. Beyond that, the creation of *librecon* and its integration into the final software image can be fully automated, so that these tasks do not result in a considerable effort for a software developer.

The automated workflow can be divided into two parts. In the first part the required compile-time information is determined by using static analysis. Here, information from the source code as well as allocation information from the final binary file have to be combined. This leads to the problem that for the generation of *librecon* the allocation in the final binary file has to be known, but *librecon* will be part of the final binary file, so that, in turn, for its creation *librecon* already has to be generated. To build *librecon* previous to the binary file, the allocation that will be made by the linker has to be predicted. Alternatively, the allocation can be computed, when *librecon* is generated, and enforced, when the binary file is created. However, for this workflow neither any assumptions concerning the used linker should be made nor should the use of a custom or customized linker be required. The purpose of this restriction is to simplify the integration of the workflow into the build process for the software of the system. Following the restriction, the effort for the design and maintenance of a custom linker can be avoided and the build process concerning the runtime components can remain unchanged. Changes of the build process for the runtime components do not affect the workflow for the generation of *librecon*. The drawback of the restriction is that no assumptions can be made about how the allocation will be computed and whether a specific allocation can be enforced. Hence, it is not possible to determine the allocation that will be made by the linker in advance. It can only be retrieved by an analysis of the created binary file.

An approach to circumvent the problem is to build the binary file initially without *librecon*, so that it can be analyzed and *librecon* can be created. Then the binary file is built once again including *librecon*. However, even if the linker is deterministic, we cannot assume that the allocation will be the same is these two cases. Since the input to the linker changes, the output and thus the allocation can differ. In this case, the information contained in *librecon* to identify data objects and to map memory addresses to objects will be invalid. Consequently, the linker must be executed only once in the build process. Hence, a way is needed to build the binary file without *librecon* and to integrate *librecon* into the binary file afterwards without changing the allocation of the data objects.

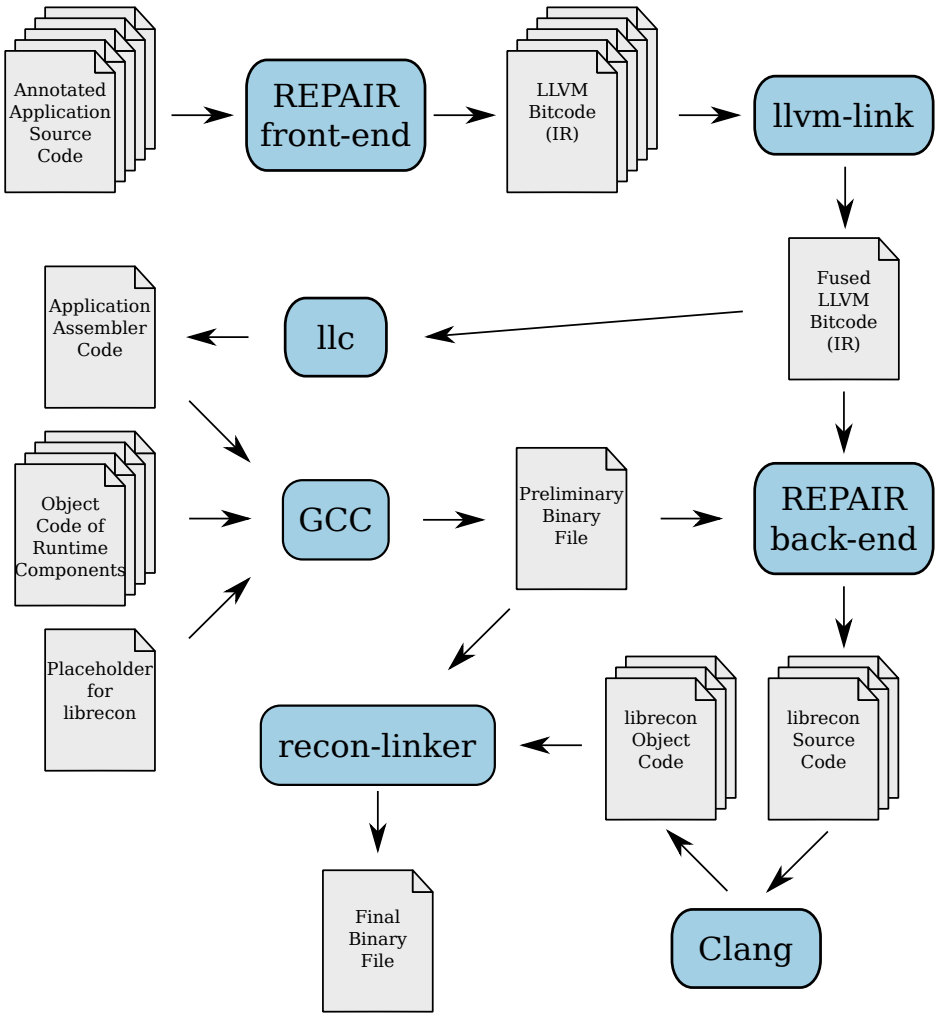

Figure 5: Automated workflow for the creation of *librecon* and its integration into the binary file

In the second part of the workflow, the collected information is encoded, and the code that processes the requests and provides the error handling information at runtime is created. Finally, *librecon* is integrated into the final binary file. The various tools and processing steps of the workflow will be presented in the following.

An overview of the complete workflow is shown in Figure 5. It starts with the processing of the annotated source code of the application. This is done by the *REPAIR front-end* which is based on the *clang* compiler front-end [5]. In the front-end, the compiler parses the source code, performs a semantic analysis, and creates an abstract syntax tree (AST), which is a high-level intermediate representation (IR) of the source code. We extended these steps for the processing of the error correction annotations, so that the syntactical correctness and semantical validity of the annotations is checked as well. Also, the AST is extended by new elements for the representation of the annotations. The last step

in the front-end is the conversion of the AST into the mid-level IR of the LLVM compiler [6]. Since this IR is independent of both source and target language, after this step no source code information is available any more. Hence, annotations in the source code must be processed before this step. This is the reason, why the evaluation of the annotations is not postponed until the allocation information is known.

Normally, a compiler continues the processing of the IR and, in the end, outputs object files. The IR is not output. However, we have to keep the IR that contains the information about the code and data of the application and the annotations, so that we can combine this information with the allocation information that we will retrieve later. Hence, we stop the processing of the compiler at this point and output the IR as LLVM bit code files. The REPAIR front-end processes each file separately. Consequently, there is an IR for each source file. In the next step we use the tool *llvm-link* [7] to fuse the set of IR to a single IR that represents the whole application. Despite its name, *llvm-link* does no allocation. It mainly replaces references to external declarations by references to the corresponding definitions if they are part of the application. This has the advantage, that all data and function definitions are available simultaneously and the annotations from different source files that refer to the same object can be checked for consistency.

Using the LLVM static compiler *llc* [8] assembly code for the target architecture is created[1] for the content of the IR. Together with the object code of the runtime components this code is input to the linker which is in our case invoked by *gcc* [9]. However, the code of the runtime components contains references to the interface functions of *librecon*. Hence, definitions of these functions are required, or otherwise the binary file cannot be created. Since to that point in time *librecon* does not exist, placeholders are defined. These placeholders reserve space for the data and code of *librecon* so that it can be integrated into the binary file without the need for a change of the allocation.

In the created binary file no relocation information is contained. Hence, it will be difficult to identify calls to the interface functions of *librecon*, since functions can be called in various ways. However, these calls have to be adapted after *librecon* has been integrated into the binary file, so that a call will transfer control properly to the start of the functions whose position is unknown when the binary is created. To solve this problem we use a trick. Functions of *librecon* can only be called by the runtime components. These are implemented in such a way that they do not contain a call to a function of *librecon*. Instead, they call trampoline functions that only contain a sequence of a few operations without effect. There is a trampoline function for each interface function of *librecon*. After *librecon* has been integrated into the binary file, the operations in the trampoline functions are replaced by a call to the corresponding function. Thus, in doing so, no identification of calls to functions of *librecon* is required. Only the position of the trampoline functions has to be determined, but these can be looked up easily in the symbol table of the binary file.

Now that the allocation has been set by the linker, this information can be extracted from the symbol table of the binary file. This takes place in the *REPAIR back-end*. It also reads the IR of the application to determine the memory layout

1 Option `-global-merge` is set to false, so that the one-to-one relation between global variables in the assembly code and the IR is preserved.

and the annotations of data objects. Next, it establishes the relation between the memory address of global variables on the one hand, and the information from the IR on the other hand. Subsequently, missing information like the relation between data objects and tasks is computed, before the resulting error handling information for data objects and the information about error correction methods is encoded as described in the previous chapter. The output is written to C source files. It represents the source code of *librecon*. By compiling the files with an arbitrary C compiler that supports the target architecture, e. g., *clang*, the runtime error classification *librecon* is created.

Finally, *librecon* has to be integrated into the binary file that has been previously created. Therefore, our tool *recon-linker* copies the content of *librecon* to the ranges that had been reserved in the binary file for this purpose. It relocates the symbols that are unresolved in the object code of *librecon* and inserts the calls to the interface function of *librecon* into the trampoline functions. These processing steps complete the workflow. Except for the recon-linker, our tools and modifications at existing tools are target independent. Thus the workflow can easily be adapted to different target architectures.

6 RELATED WORK

Except for the use case presented in this report, compiler information that is required at runtime is typically used for debugging. For this purpose, the DWARF format [10] has become established. The current DWARF version 4 standard is supported by several compilers, linkers and debugging tools. Providing compile-time information at runtime in this format would follow an approved approach. The main objective of DWARF is flexibility, so that it is usable for a variety of source and target languages. By adding new attributes for data objects and functions that are correction methods, the debug information can be extended by the required error handling information. However, the DWARF format is not efficient for passing error handling information in resource constrained systems.

On the one hand the extension of debug information would not be memory-efficient. Most of the information that is useful for debugging is not relevant for error correction. This includes variable names, type qualifiers, type names, and references to the source code. Even if all this information would be omitted, the way information is stored causes unwanted overhead. In DWARF format, information is stored as a series of key-value pairs. To save memory, recurring sequences of keys can be stored in a directory like manner. Nevertheless, for our approach the flexibility given by the attribute keys is not needed at all. Since the kind of error handling information that is provided for data objects or for error correction methods is the same for all data objects or all methods, respectively, the sequence of values can be fixed and need not be encoded by the use of attribute keys.

On the other hand a special format for the encoding of error handling information can be more runtime efficient than the DWARF format. The fixed structure of the entries that store the error handling information enables direct accesses on the desired information. In contrast, the encoding as key-value pairs would require a search for the matching key. Anyway, the efficient evaluation of the information stored in the DWARF format is up to the debugging tools. E. g., to determine the data object that is allocated to a given memory address, a search for the debug information entry of the data object is required. Consequently several debug information entries have to be processed and the sequence of operations that describe the memory location of a data object has to be evaluated. Fortunately, for global objects this sequence consists of only one operation. Nevertheless, there is no mapping of memory addresses to objects[1]. Such a mapping has to be created by the debugger, that has to read the complete debug information for data objects beforehand, or the DWARF information has to be searched for each request. Our runtime error classification contains information that is already preprocessed for efficient search operations and also provides the methods for the efficient evaluation of queries.

1 The table `.debug_aranges` accelerates the lookup of data objects by address by mapping memory ranges to compilation units that contain the desired entry. Still the compilation unit has to be browsed for the debug information entry of the data object.

Finally, the DWARF format does not efficiently support the encoding of distinct information for different nested objects that result from the same definition in the source code. Debug information for nested objects in two different global variables with the identical type will be the same for the nested objects with the same offset within the composite objects. But this need not be true for error handling information, since the use and semantics of the nested objects can be different. In the DWARF format, information about nested objects is directly linked to the description of the containing data type. To encode distinct information for a nested object depending on the enclosing object it belongs to, a separate description of the data type for each enclosing object would be needed. Hence, the description of a composite data types cannot be used for all objects of that type.

7 CONCLUSIONS

In this report we presented our approach for passing error handling information that is computed at compile-time to components that use this information at runtime. Central component of the approach is the library for runtime error classification *librecon* that contains the error handling information. We described the possible queries for error handling information by the runtime components in three use cases and presented an implementation that supports these use cases efficiently. Moreover, the requirements of embedded systems are taken into account by avoiding the storage of redundant information and enabling fast accesses to the information. Finally we showed, how the runtime error classification can be generated application-specific in an automated workflow.

The intention of the presented approach is to prove the feasibility of passing the information in an efficient way. Still, there is room for improvements. One option is to reduce the memory footprint needed by *librecon*. Since the runtime components access the error handling information without knowledge about the encoding of the information, this encoding can be replaced. E. g., compression techniques can be applied. However, the effect of a compression on the response time has to be considered, since also a decompression of the data has to take place, before data can be retrieved. A compression can have a negative impact on the response time, since the decompression requires computation time, but it can also have a positive effect on the amount of memory transfers needed to load the desired error handling information. As another side effect, during decompression the uncompressed data has to be stored which increases the usage of temporal memory. This increases the likelihood, that an error affects *librecon*. All in all, a thorough analysis of the trade-offs is required for designing an improved version of *librecon*.

BIBLIOGRAPHY

[1] Florian Schmoll, Andreas Heinig, Peter Marwedel, and Michael Engel. Improving the fault resilience of an H.264 decoder using static analysis methods. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(1s):31:1–31:27, December 2013.

[2] Peter Marwedel. *Embedded Systems Design - Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2nd edition, 2011. ISBN 978-94-007-0256-1.

[3] ITRS. International Technology Roadmap for Semiconductors, 2013 Edition, Process Integration, Devices, and Structures (PIDS). http://www.itrs.net/Links/2013ITRS/Home2013.htm.

[4] 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.

[5] clang: A C language family frontend for LLVM. http://clang.llvm.org/.

[6] LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html.

[7] llvm-link - LLVM bitcode linker. http://llvm.org/docs/CommandGuide/llvm-link.html.

[8] llc - LLVM static compiler. http://llvm.org/docs/CommandGuide/llc.html.

[9] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[10] DWARF Debugging Information Format Committee. DWARF Debugging Information Format, Version 4, 2010. http://www.dwarfstd.org/Dwarf4Std.php.