An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs

Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. Runtime assertion checking provides a powerful, highly automatizable technique to detect violations of specified program properties. However, monitoring of annotations for pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc.) is not straightforward and requires systematic instrumentation and monitoring of memory-related operations.
This paper describes the runtime memory monitoring library we developed for execution support of E-ACSL, executable specification language for C programs offered by the FRAMA-C platform for analysis of C code. We present the global architecture of our solution as well as various optimizations we realized to make memory monitoring more efficient. Our experiments confirm the benefits of these optimizations and illustrate the bug detection potential of runtime assertion checking with E-ACSL.
Keywords: runtime assertion checking, memory monitoring, executable specification, invalid pointers, memory-related errors, FRAMA-C, E-ACSL.

1 Introduction

Memory related errors, including invalid pointers, out-of-bounds memory accesses, uninitialized variables and memory leaks, are very common. For example, the study for IBM MVS software in [1] reports that about 50% of detected software errors were related to pointers and array accesses. This is particularly an issue for a programming language like C that is paradoxically both the most commonly used for development of system software with various critical components, and one of the most poorly equipped with adequate protection mechanisms. The C developer is responsible for correct allocation and deallocation of memory, pointer dereferencing and manipulation (like casts, offsets, etc.), as well as for the validity of indices in array accesses.

Among the most useful techniques for detecting and locating software errors, *runtime assertion checking* is now a widely used programming practice [2]. Turing advocated the use of assertions already in 1949 and wrote that "the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows" [3]. A lot of research works have addressed efficient techniques and tools for runtime assertion checking. Leucker and Schallhart provide a survey on *runtime*

verification and conclude that "one of its main technical challenges is the synthesis of efficient monitors from logical specifications" [4]. An efficient memory monitoring for C programs is the purpose of the present work.

In this paper, we present the solution for memory monitoring of C programs we have developed for runtime assertion checking in Frama-C [5], a platform for analysis of C code. It includes an expressive executable specification language E-ACSL and a translator, called E-ACSL2C in this paper, that automatically translates an E-ACSL specification into C code [6, 7]. In order to support memory-related annotations for pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc.), we need to keep track of relevant memory operations previously executed by the program. Hence, we have developed a monitoring library for recording and retrieving validity and initialization information for the program's memory locations, as well as an automatic instrumentation of source code in E-ACSL2C inserting necessary calls to the library during the translation of an E-ACSL specification into C.

The proposed solution is designed both for *passive* and *active* monitoring, though this paper discusses only passive monitoring, that is the default one. Passive monitoring only aims at observing and reporting failures, while active monitoring introduces new actions e.g. for recovery from detected erroneous situations. Our solution implements a *non-invasive* source code instrumentation, that is, monitoring routines do not change the observed behavior of the program. In particular, it does not modify the memory layout and size of variables and memory blocks already present in the original program, and may only record additional monitoring data in a separate memory store.

The contributions of this paper include:

- a detailed description of our solution of memory monitoring for runtime assertion checking with Frama-C [5], allowing to automatically generate monitors from assertions and function contracts written in the E-ACSL specification language [6];
- an efficient storage of memory related operations based on Patricia tries [8];
- optimized records and queries in the store for faster recording and retrieving information on memory blocks;
- an optimized instrumentation reducing the amount of memory monitoring for memory locations that are irrelevant with respect to the provided assertions;
- experiments illustrating the benefits of these optimizations and the capacity of error detection using E-ACSL.

The paper is organized as follows. Sec. 2 presents the context of this work, including Frama-C and E-ACSL. Sec. 3 gives a global overview of our solution for memory monitoring, in particular, the instrumentation realized by E-ACSL2C and the basic primitives provided by our monitoring library. Optimized data storage and search operations are described respectively in Sec. 4 and 5. Sec. 6 presents the optimization reducing irrelevant memory monitoring. Our initial experiments are described in Sec. 7 and summarized at the end of Sec. 4, 5 and 6. Finally, Sec. 8 and 9 present respectively related work and the conclusion.

E-ACSL keyword	Its semantics
`\base_addr(p)`	the base address of the block containing pointer `p`
`\block_length(p)`	the size (in bytes) of the block containing pointer `p`
`\offset(p)`	the offset (in bytes) of `p` in its block (i.e., w.r.t. `\base_addr(p)`)
`\valid_read(p)`	is true iff reading `*p` is safe
`\valid(p)`	is true iff reading and writing `*p` is safe
`\initialized(p)`	is true iff `*p` has been initialized

(here `p` must be a non-void pointer)

Fig. 1: Memory-related E-ACSL constructs currently supported by E-ACSL2C.

2 Executable specifications require memory monitoring

The executable specification language E-ACSL [6, 9] was designed to support runtime assertion checking in Frama-C. Frama-C [5] is a platform dedicated to analysis of C programs that includes various analyzers, such as abstract interpretation based value analysis (Value plug-in), dependency analysis, program slicing, Jessie and WP plug-ins for proof of programs, etc. ACSL [10] is a behavioral specification language shared by different Frama-C analyzers that takes the best of the specification languages of earlier tools Caveat [11] and Caduceus [12], themselves inspired by JML [13].

ACSL is expressive enough to express most functional properties of C programs and has already been used in many projects, including large-scale industrial ones [5]. It is based on a typed first-order logic in which terms may contain *pure* (*i.e.* side-effect free) C expressions and special keywords. An Eiffel-like contract [14] may be associated to each function in order to specify its pre- and postconditions. The contract can be split into several named guarded behaviors. Contracts may also be associated to statements, as well as assertions, loop invariants and loop variants. ACSL annotations also include definitions of (inductive) predicates, axiomatics, lemmas, logic functions, data invariants and ghost code.

Designed as a large subset of ACSL, E-ACSL preserves ACSL semantics. Moreover, the E-ACSL language is *executable*: its annotations can be translated into C monitors by E-ACSL2C and executed at runtime. This makes it suitable for runtime assertion checking. Fig. 1 presents some memory-related E-ACSL annotations. We use the term *(memory) block* for any (statically, dynamically or automatically) allocated object. A block is characterized by its size and its *base address,* that is, the address of its first byte.

Fig. 2 shows a simple C function `findchr` with an ACSL contract (that is also an E-ACSL contract) enclosed into `@`-comments. Given a character `c` and a pointer `s` to an array of `n` characters, `findchr` returns a pointer to an occurrence of `c` in the array, and `NULL` otherwise. It is very similar to the C standard `memchr` function (basically, our contract does not require to find the *first* occurrence of `c`). The contract contains two behaviors (lines 2–6, 7–9) with a common precondition (line 1). The precondition states that `s` must refer to a valid readable location with at least `n` characters to the right of `s`. The first behavior `found` is defined by the **`assumes`** clause line 3. Whenever the **`assumes`** condition is satisfied, the behavior's postconditions (lines 4–6) must be ensured. They state that the returned

```
1  /*@ requires \valid_read(s) && \offset(s)+n <= \block_length(s);
2    @ behavior found:
3    @   assumes \exists int i; 0 <= i < n && s[i] == c;
4    @   ensures \base_addr(s) == \base_addr(\result);
5    @   ensures \offset(s) <= \offset(\result) < \offset(s)+n;
6    @   ensures * \result == c;
7    @ behavior not_found:
8    @   assumes \forall int i; 0 <= i < n ==> s[i] != c;
9    @   ensures \result == \null;
10   @*/
11 char * findchr(char *s, char c, unsigned int n) {
12   unsigned int i;
13   for(i = 0; i < n; i++)
14     if(s[i] == c)
15       return s+i;  // found, returns the pointer
16   return (void*)0; // not found, returns NULL
17 }
```

Fig. 2: Function `findchr` specified with an E-ACSL contract.

value (keyword `\result`) must refer to the same block as `s` (line 4), to one of the `n` characters starting from `*s` (line 5), and the referred character must be equal to `c` (line 6). Similarly, the second behavior states that the null pointer must be returned (line 9) whenever `c` in not present in the array (line 8).

Translation into C of basic E-ACSL features (including overflow-free arithmetic operations for integers, behaviors, quantifications over finite sets, some special keywords, values at the Pre, Post or any labeled state, etc.) was described in [6]. However, runtime assertion checking of E-ACSL specifications involving memory-related constructs of Fig. 1 is particularly complex. Languages with pointers, such as C or C++, do not allow the developer to easily check for pointer validity. The developer is supposed to know when a pointer is valid or not. For example, even when the size of an input array `a` is provided in a function signature `int f(int a[10])`, it is ignored according to the ISO C 99 norm [15, Sec. 6.7.5.3.7]. In other words, this declaration is equivalent to `int f(int *a)`, so the array size is lost. At runtime, `sizeof(a)` inside `f` returns the size of a pointer, and nothing guarantees that `a` really refers to an array with 10 elements. Runtime checking of memory-related E-ACSL annotations can be realized using systematic monitoring of memory operations as shown in the next section.

3 Memory monitoring for E-ACSL: an overview

Runtime assertion checking of E-ACSL specifications is based on a non-invasive source code instrumentation by E-ACSL2C. In order to evaluate memory-related E-ACSL annotations (Fig. 1), we record information on validity and initialization of memory locations during program execution in a dedicated data store, that we call below *the store.* We have developed a memory monitoring library that provides primitives for both evaluating memory-related E-ACSL annotations (by making queries to the store) and recording in the store all necessary data on allocation, deallocation and initialization of memory blocks. Thus E-ACSL2C inserts calls to library primitives for two purposes:

– to translate into C and evaluate memory-related E-ACSL annotations; and
– to record memory-related program operations in the store.

The following subsections present these two aspects in detail.

3.1 Translation and evaluation of memory-related annotations

When a specified property is violated at runtime, the instrumented code generated by E-ACSL2C calls a special function, that we denote here `e_ascl_fail`, whose default version reports the assertion failure and exits the execution.[1]

The instrumentation is different for an internal annotation in a function and for a function contract. An annotation inside a function `f` is directly translated by E-ACSL2C into C code that checks the annotation condition inside `f` (this case will be illustrated on Fig. 7). For a function `f` with a function contract, E-ACSL2C adds a new C function `__e_acsl_f` with the same signature as `f`, and replaces all initial calls to `f` by calls to `__e_acsl_f`. Basically, `__e_acsl_f` contains three parts: checking the precondition of `f`, a call to `f` and checking the postcondition of `f`. Thus a contract of `f` is systematically checked by `__e_acsl_f` in the instrumented code whenever `f` is called in the original code, even if the code of `f` is not provided during the instrumentation step.

The library provides primitives for the most frequently used memory-related E-ACSL annotations shown in Fig. 1. They have the same role and similar names (with "__" as prefix instead of "\"). For the last three of them, library functions expect an additional second argument indicating the size (in bytes) of the memory location `*p`.

For example, Fig. 3 presents (a simplified version of) function `__e_acsl_findchr` automatically generated by E-ACSL2C for the function `findchr` of Fig. 2. Lines 4–5 of Fig. 3 check the precondition (and report any violation). Lines 6–9 compute if the first behavior's **`assumes`** clause is satisfied, i.e. if this behavior is applicable for the current call of `findchr`. Since execution of an E-ACSL annotation must not introduce any risk of runtime error, an additional check of validity of reading `s[i]` is automatically added at line 7 before an access to `s[i]`. Similarly, lines 10–13 compute if the second behavior is activated by the current call of `findchr`. Then `findchr` is called line 14. Next, lines 15–22 check that if the first behavior's assumption is true, its postconditions are satisfied as well. Again, to avoid any risk of a runtime error inserted by E-ACSL2C, an additional validity check is added at line 20 before an access to `*__res`. Similarly, the second behavior is checked at lines 23–25.

3.2 Recording validity and initialization data in the store

In order to be able to provide requested information on memory locations, the code instrumented by E-ACSL2C records in the store for each block the *block*

[1] Actual instrumentation allows the user to customize this function by defining its own action according to several parameters [7].

```
1  char * __e_acsl_findchr(char *s, char c, unsigned int n) {
2    char * __res; unsigned int i;
3    int __e_acsl_exists = 0; int __e_acsl_forall = 1;
4    if(!( __valid_read(s,1) && __offset(s)+n <= __block_length(s) ))
5      e_acsl_fail("findchr","Pre","line␣3");
6    for( i=0; i<n && __e_acsl_exists == 0 ; i++ ) {
7      if(! __valid_read(s+i,1)) e_acsl_fail("findchr","mem_access:s[i]","line␣5");
8      if( s[i] == c ) __e_acsl_exists = 1;
9    }
10   for( i=0; i<n && __e_acsl_forall == 1 ; i++ ) {
11     if(! __valid_read(s+i,1)) e_acsl_fail("findchr","mem_access:s[i]","line␣10");
12     if(!( s[i] != c )) __e_acsl_forall = 0;
13   }
14   __res = findchr(s, c, n);
15   if( __e_acsl_exists ){
16     if(!( __base_addr(s) == __base_addr(__res) ))
17       e_acsl_fail("findchr","Post","line␣6");
18     if(!( __offset(s) <= __offset(__res) && __offset(__res) < __offset(s)+n ))
19       e_acsl_fail("findchr","Post","line␣7");
20     if(! __valid_read(__res,1)) e_acsl_fail("Post","mem_access:*__res","line␣8");
21     if(!( *__res == c )) e_acsl_fail("Post","findchr","line␣8");
22   }
23   if( __e_acsl_forall )
24     if(!( __res == (void *)0 )) e_acsl_fail("Post","findchr","line␣11");
25   return __res;
26 }
```

Fig. 3: Simplified version of function `__e_acsl_findchr` automatically generated by E-ACSL2C for runtime checking of the contract for the function `findchr` of Fig. 2.

Function	Its meaning
`__store_block(p,len)`	records a block of size `len` and base address `p` in the store
`__delete_block(p)`	removes existing block with base address `p` from the store
`__malloc(len)`	allocates a block of size `len` and records it in the store
`__free(p)`	deallocates the block with base `p` and removes it from the store
`__initialize(p,len)`	marks `len` bytes starting from pointer `p` as initialized
`__full_init(p)`	marks the whole block with base address `p` as initialized

Fig. 4: Basic recording primitives provided by the memory monitoring library.

metadata including its base address, size (in bytes), validity status (whether reading or writing the block is safe) and the initialization status for each byte of the block. Our memory monitoring library has been designed to be compatible with various implementations of the underlying store. This section presents the instrumentation scheme for recording operations using high-level library primitives, while the store implementation and optimizations are discussed later in Sec. 4, 5 and 6.

Fig. 4 presents some recording primitives provided by the library. They allow to register a new block in the store, to mark some particular bytes (or the whole block) as initialized and to remove the block from the store when it is not valid anymore. For convenience, instrumented versions of basic dynamic allocation functions (`malloc`, `calloc`, `realloc`, `free`) are provided as well. They directly add/remove from the store the allocated/deallocated block (and, in case of `realloc`, transfer recorded initialization information for the old block to the new one).

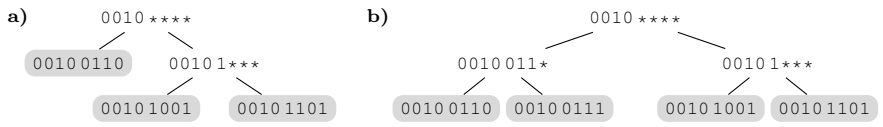

Fig. 5: Example of a Patricia trie **a)** before, and **b)** after inserting `0010 0111`

Thanks to these primitives, the (unoptimized) instrumentation for recording in the store is mostly straightforward. To monitor the block of an argument or a local variable `v` of type `T` in function `f`, E-ACSL2C adds the calls `__store_block(&v,sizeof(T))` in the beginning, and `__delete_block(&v)` at the end of the scope of `v`. For a global variable, these calls are inserted in the beginning and at the end of the function `main`. In addition to them, for global variables (initialized by default to 0 in C) and function arguments (initialized by a function call), the `__store_block(&v,sizeof(T))` is followed by `__full_init(&v)` to mark the whole block as initialized. To monitor an assignment `v = exp;` to a variable (or a left value) `v`, a call to `__initialize(&v,sizeof(v));` is inserted. Literal strings and initializers are easily handled as well. Finally, dynamic allocation functions are simply replaced by their instrumented counterparts.

4 Optimized storage for the memory monitoring library

Efficient implementation of the store requires a data structure with a good time and space complexity, since the instrumented code may perform frequent modifications and lookups in the store. It is intuitively clear that the structure has to be sorted: treating E-ACSL constructs may require to access a block metadata directly by its base address as well as to find a block's predecessor or successor. For example, the query `__base_addr(p)` searches the store for the closest to `p` base address less than `p` (and checks the bounds afterwards). Thus, a hash table will not fit. Lists are not efficient enough due to a linear worst-case complexity. Unbalanced binary search trees have a linear worst-case complexity too when inserted base addresses are monotonically increasing, and this may be quite common. Finally, the cost of balancing (e.g. in a self-balancing binary search tree) would be amortized if the store modifications (that may lead to rebalancing) were less frequent than simple queries (that take advantage of a balanced structure). For tested examples of code instrumented by E-ACSL2C this is not necessarily true.

Our implementation of the store is based on a *Patricia trie* [8], also known as a *radix tree* or *compact prefix tree*, which is efficient even if the tree is unbalanced. Node keys are base addresses (e.g. 32-bit or 64-bit words) or address prefixes. Any leaf contains a block metadata with the block base address. Routing from the root to a block metadata is ensured by internal nodes, each of them contains the greatest common prefix of base addresses stored in its successors.

For instance, Fig. 5a shows a Patricia trie, for simplicity, over 8-bit addresses. It contains three blocks in its leaves (only block base addresses are shown here), and greatest common prefixes in internal nodes. A "$\star$" denotes meaningless bits

following the greatest common prefix. Fig. 5b presents another trie obtained from the first one by adding the base address `0010 0111`, that required to create a new internal node `0010 011*`. Conversely, removing `0010 0111` from the trie of Fig. 5b would give that of Fig. 5a.

Theoretical worst-case complexity of a lookup in a Patricia trie in our case is $O(k)$ where k is the word length (e.g. 32-bit or 64-bit). In practice, since a program is allowed to allocate blocks in a limited memory space, the trie height remains far below this upper bound. In addition, unlike for arbitrary strings, comparisons for words can be very efficiently implemented by bit operations (see also Sec. 5).

Storage of a block metadata takes a few bytes, except for initialization information when the block itself is long. In this case, initialization of each byte is monitored separately (bit-level initialization through bit-fields is not yet supported). To reduce the memory space occupied by the store, recording block initialization information is optimized in two ways. First, since initialization of each byte in a block can be recorded in one bit, block initialization is recorded in a dynamically allocated array, whose size is therefore 8 times less than the block size. Second, when none or all of the bytes of a memory block have been initialized (that are very common cases), initialization array is freed. Instead, an integer field counting initialized bytes is used. Third, the `__full_init` primitive can be used to mark the whole block as initialized, avoiding multiple calls to `__initialize` for particular bytes.

Experiments. To choose which datastructure is most appropriate for implementing the store, we compared the implementation based on Patricia tries to three other implementations of the store: based on linked lists, on unbalanced binary search trees, and on Splay trees used in earlier memory safety related tools (see e.g. [16]). Our implementation appears to be in average more than 2500 times faster than linked lists, 200 times faster than unbalanced binary search trees and 27 times faster than Splay trees. For linked lists and search trees, it confirms the intuition given earlier in this section. The results for Splay trees are comparable to Patricia tries on most examples, maybe 2-3 times faster for some examples, and dramatically (> 500 times) slower for examples (like multiplication of big matrices, matrix inversion etc.) where the program's consecutive accesses to memory are not at all performed to the same memory blocks. The reason is also intuitively clear: since Splay trees move recently accessed elements to the top of the tree, this takes time and brings no benefit when the following queries to the store are not related to the same memory blocks again. For instance, since matrix multiplication requires to take elements in different rows and columns each time, multiplication of big matrices, where all matrix elements do not fit to the same memory block, results in loss of time due to useless restructuring of the Splay tree. On the contrary, on programs with frequent consecutive accesses to the same block metadata in the store, Splay trees appear to be (up to three times in our examples) more efficient.

```
1  typedef unsigned char byte;
2  // index            0    1    2    3    4    5    6    7    8
3  byte  masks[] = {0x00,0x80,0xC0,0xE0,0xF0,0xF8,0xFC,0xFE,0xFF};
4  int longer [] = {   0,  -1,   3,  -3,   6,  -5,   7,   8,  -8};
5  int shorter[] = {   0,   0,   1,  -2,   2,  -4,   5,  -6,  -7};
6  byte gtCommonPrefixMask(byte a, byte b) {
7    byte nxor = ~(a ^ b);    // a bit = 1 iff this bit is equal in a and b
8    int i = 4;               // search starts in the middle of the word
9    while(i > 0)             // if more comparisons needed
10     if (nxor >= masks[i]) i = longer[i]; // first i bits equal,try a longer prefix
11     else i = shorter[i];                 // otherwise, try a shorter prefix
12   return masks[-i];        // if i<=0, masks[-i] is the answer
13 }
```

Fig. 6: Search for greatest common prefix mask, illustrated here for bytes

5 Optimized records and queries in the store

Queries for adding, removing, or searching a given base address A in the store based on a Patricia trie require comparisons of A with existing nodes and computations of the greatest common prefix for two elements (cf Fig.5). For Patricia tries storing addresses (strings of 0's and 1's of fixed length rather than strings of arbitrary length over a wider set of characters), these comparisons may be simplified due to the nature of elements. Let us call by the *greatest common prefix mask* M of A and B the mask containing 1's for the positions of common bits in the greatest common prefix P of A and B. So M starts with n 1's followed by 0's, where n is the number of common bits in P. For example, the greatest common prefix of bytes $A =$0110 0111 and $B =$0111 1111 is $P =$111$\star$ $\star\star\star\star$, while the greatest common prefix mask is $M =$1110 0000.

We carefully optimized all prefix computations and comparisons by intensive usage of efficient bit-to-bit operations. Interestingly, one optimization that we realized for computation of the greatest common prefix mask appeared particularly efficient. Fig. 6 illustrates the optimized version, for simplicity, over bytes instead of words. It is based on the classic dichotomic search of the index `i` such that the greatest common prefix mask starts with exactly `i` 1's. In addition to precomputed masks (line 3) and bit operations (line 7), our version uses precomputed indices (lines 4,5) for the next prefix length `i` to try, therefore it avoids the usual `mid=(high+low)/2` computation at each iteration, making frequent calls to the function much faster. A negative value `i<=0` indicates that `-i` is the final greatest common prefix length. The next value of `i` is simply extracted (lines 10,11) of the arrays depending if the next candidate prefix should be tried longer or shorter. For instance, for A and B above, `nxor` equals the byte `11100111`, and the function will try `i=4`, then `i=shorter[4]=2`, then `i=longer[2]=3` and finally stop with `i=longer[3]=-3` and return the mask `masks[3]=0xE0` of length 3, that is in binary precisely `1110 0000`.

Experiments. We compared our optimized implementation to a non-optimized version of the common prefix mask computation based on the usual comparison of strings commonly used for Patricia tries (with a linear run over the elements from left to right, that we have also optimized by bit operations). On the tested examples, our optimized version illustrated by Fig. 6 makes the execution of the

```
1  #include<stdlib.h>
2  int last;
3  int* new_inversed(int len, int *v) {
4    int i, *p;
5    //@ assert \valid(v) && \offset(v)+len*sizeof(int) <= \block_length(v);
6    p = malloc(sizeof(int)*len);   // allocate a new vector p
7    for(i=0; i<len; i++)
8      p[i] = v[len-i-1];           // write inversed vector v into p
9    return p;
10 }
11 void main() {
12   int v1[3]={1,2,3}, *v2;
13   //@ assert \valid(&v1[2]);
14   last = v1[2];
15   v2 = new_inversed(3, v1);
16   last = v2[2];
17   //@ assert last == 1;
18   free(v2);
19 }
```

Fig. 7: File `vector.c` where the function `new_inversed` allocates and returns a new vector containing the inversed given vector `v` of `len` integers.

instrumented code in average 2.7 times faster. This rate goes up to 4.7 times for examples with intensive usage of the memory monitoring library.

6 Optimized instrumentation using static analysis

The instrumentation presented in Sec. 3 is sound and complete: the code instrumented by E-ACSL2C reports an E-ACSL annotation failure at runtime if and only if this E-ACSL annotation is indeed violated. However it has the major drawback of being hugely verbose and time-consuming: for each variable, each (de)allocation and each assignment, one or even several new statements are generated. It is however sufficient to monitor the memory locations involved in memory-related constructs in the provided E-ACSL annotations.

To solve this drawback, we have designed an interprocedural backward dataflow analysis which computes an over-approximated set σ of memory locations that it is sufficient to monitor in order to preserve soundness and completeness of the instrumentation. Let us explain on the example of Fig. 7 how this analysis works (for lack of space, we do not give its formal presentation here). Without any analysis, we have to monitor every variable of the program and to record when it is allocated, initialized and deallocated by systematically adding calls to the recording primitives of Fig. 4 as explained in Sec. 3.2.

However, this monitoring is only required for memory blocks involved in memory-related E-ACSL constructs. In our example, they are `\valid(v)`, `\offset(v)` and `\block_length(v)` at line 5, and `\valid(&v1[2])` at line 13. So we need to monitor the formal parameter `v` of function `new_inversed` and the location `&v1[2]`. For the latter, we keep less precise information. Our current analysis is purely syntactical and does not perform any precise semantic aliasing analysis. To be sound, we perform an over-approximation and monitor any information about the whole local array `v1` of function `main`, including `*(v1+i)` for any offset `i`. Basically, from

```
1  int last;
2  int* new_inversed(int len, int *v) {
3    int i, *p;
4    __store_block(&v,sizeof(int*)); __full_init(&v);
5    if(!( __valid(v,sizeof(int)) && __offset(v)+len*sizeof(int) <= __block_length(v) ))
6      e_acsl_fail("new_inversed","assert","line␣4");
7    p = __malloc(sizeof(int)*len);
8    for(i=0; i<len; i++)
9      p[i] = v[len-i-1];
10   __delete_block(&v);
11   return p;
12 }
13 int main() {
14   int v1[3]={1,2,3}, *v2;
15   __store_block(v1,3*sizeof(int)); __full_init(v1);
16   if(! __valid(v1+2,sizeof(int)) ) e_acsl_fail("main","assert","line␣13");
17   last = v1[2];
18   v2 = new_inversed(3, v1);
19   last = v2[2];
20   if(!( last == 1 )) e_acsl_fail("main","assert","line␣17");
21   __free(v2);
22   __delete_block(v1); __clean();
23 }
```

Fig. 8: Simplified instrumentation of the file `vector.c` of Fig. 7 with E-ACSL2C.

these E-ACSL annotations, the analysis goes backwards in the code in order to find where the monitored variables `v` and `v1` are assigned and where aliases are potentially created.

More precisely, the analysis starts from the end of the program with $\sigma = \emptyset$, and goes backwards up to the beginning, analyzing statements, annotations and called functions in order to collect memory locations to be monitored into σ. For the example of Fig. 7, it collects nothing until the assertion at line 5 in function `new_inversed` called from the line 15 (still in a context with $\sigma = \emptyset$). At this point, it remembers that `v` has to be monitored. Going back to the callsite (line 15), as the formal parameter `v` has to be monitored, the corresponding argument `v1` is also collected into σ. For the assertion of line 13, the analysis computes that `v1` has to be monitored (actually, it is already in σ, so nothing new is discovered). Finally the analysis concludes that `v` and `v1` have to be monitored, leading to the optimized instrumentation of Fig 8. We notice that variables `last`, `len`, `i`, `p` and `v2` are not monitored, unlike in the unoptimized instrumentation.

If `v1` was a pointer referring to another array `v3` (e.g. if the line 12 was `int v3[3]={1,2,3}, *v1=v3, *v2;`), the analysis would deduce from the assignment `v1=v3` that `v3` should be monitored as well.

Experiments. In the tested examples, the optimization based on dataflow analysis reduced the total execution time of instrumented code by 66% in average. It is due to a smaller number of monitored variables (decreasing by 78% in average) and hence a smaller number of records and queries to the store (number of calls to `gtCommonPrefixMask` throughout the execution decreased by 71% in average). The analysis was rather fast: no example has been slowed down by integrating this optimization.

Example	Orig.	Lists	BST	PT−anal.	PT−mask	**PT**	Splay	Valgrind
binarySearch	0.01	0.51	0.62	1.59	0.53	0.53	0.64	0.27
insertionSort	0.12	1.27	1.26	3.86	1.25	1.25	1.30	2.81
matrixMult	0.01	58.48	90.43	9.01	8.57	8.75	398.60	0.48
matrixInv	0.02	21.54	29.94	1.90	1.42	1.53	145.80	0.47
quickSort	0.01	11.15	2.67	0.48	0.36	0.13	0.02	0.27
bubbleSort	0.22	4.64	7.16	32.58	7.26	6.90	7.21	3.36
merge	0.01	101.33	94.80	0.29	0.47	0.14	0.05	0.45
RedBlackTree	0.01	101.69	145.20	0.30	0.39	0.27	19.59	0.51
mergeSort	0.01	>24h	>24h	513.85	25.02	7.63	2.50	0.27

Fig. 9: Detailed execution time (in sec.) for selected examples and techniques.

7 Experimental results

Performances. To evaluate our memory monitoring solution, we performed in total more than 300 executions for more than 30 programs obtained from about 10 examples with different levels of specifications and values of parameters. These initial experiments were conducted on small-size examples because they were mostly manually specified in E-ACSL. We measured the execution time of the original code and of the code instrumented by E-ACSL2C with various options in order to evaluate their performances (with and without optimizations, with four different implementations of the store, etc.). Such indicators as the number of monitored variables, memory allocations, records and queries in a Patricia trie were recorded as well.

Fig. 9 presents some of these examples and indicators in detail. Its columns give execution time of the original program, using store implementation based on lists, binary search trees, three versions of Patricia tries and Splay trees. Patricia tries based implementations were tested respectively without dataflow analysis of Sec. 6, without query optimization of Sec. 5 and with both optimizations. Time of analysis with Valgrind tool [17] is indicated in the last column.

Most experimental results have already been summarized at the end of Sec. 4, 5 and 6, where the average rates were computed for a complete list of examples (some of which are not presented in Fig. 9). We notice in addition that the execution time with Valgrind is not comparable with our solution and may depend on the number of memory-related annotations in the specification.

Error detection capacity. In addition to performance evaluation, we used mutational testing to evaluate the capacity of error detection using runtime assertion checking with FRAMA-C. We considered five annotated examples (see Fig. 10), generated their *mutants* (by performing a *mutation* in the source code) and applied assertion checking on them. Annotations included preconditions, postconditions, assertions, memory-related constructs etc., and were written in E-ACSL. Mutations included: numerical-arithmetic operator modifications, pointer-arithmetic operator modifications, comparison operator modifications and logic (*land* and *lor*) operator modifications. The PATHCRAWLER test generation tool [18] has been used to produce test cases. Each mutant was

	alarms	mutants	equivalent	killed	% erroneous killed
fibonacci	19	27	2	25	100%
bubbleSort	15	44	2	42	100%
insertionSort	10	39	3	36	100%
binarySearch	7	38	1	37	100%
merge	5	92	5	87	100%

Fig. 10: Error detection for mutants

instrumented by E-ACSL2C and executed on each test case in order to check at runtime if the specification was satisfied. The original programs successfully passed all these checks. As usual, when a violation of an annotation was reported for at least one test case, the mutant was considered to be *killed*. Fig. 10 illustrates the results. Except for equivalent mutants (where the mutation produced by chance an equivalent program), all erroneous mutants were killed.

8 Related work

Runtime assertion checking. The present work is part of an extension of FRAMA-C, an existing toolset for analysis of C code, for supporting runtime assertion checking. It is therefore related to a lot of works on runtime assertion checking [2] and, more generally, runtime verification [4]. More specifically, one of our main objectives is to support and execute annotations in E-ACSL, an expressive executable specification language shared by static and dynamic analysis tools. Hence, our work continues previous contributions to development of expressive specification languages such as Eiffel [14], JML [19] for Java and Alfa [20] for Ada.

Memory safety. Since the main purpose of this paper is the support of memory-related E-ACSL annotations, our work is also related to previous efforts for ensuring memory safety of C programs at runtime. They include safe dialects of C, specific fail-safe C compilers and memory safety verification tools for C code. In particular, the idea to store object metadata on valid memory blocks in a separate database was previously exploited in [16, 21–24] and appeared well-adapted for most spatial errors (that is, accesses outside the bounds [25]). Advantages of this approach include relative efficiency (propagation of pointer metadata at each pointer assignment is not required) and compatibility (the memory layout of objects is preserved). However, this technique results in significant time overhead due to lookup operations in the database, and is not directly adapted to detect sub-object overflows inside nested objects (e.g. an array of structures) and temporal errors (that is, accesses to an object that has been deallocated [25]). An alternative approach is based on pointer metadata stored inside multi-word *fat pointers* extending the pointer representation with bounds information [26–28]. Hybrid techniques combining ideas of both approaches have been proposed as well [29, 25]. The technique of *shadow pages* [17, 30] makes it possible to immediately find stored validity information for a pointer without providing an easy

way to find the base address of the block, block size and pointer offset required by memory-related E-ACSL clauses.

Our global objective is quite different from these efforts. Unlike these advanced works focused on detection of memory safety errors, we aim at supporting runtime checking for memory-related annotations of an expressive specification language E-ACSL. Even if we have already realized several optimizations, performances of our implementation remain below the most advanced proposals addressing memory safety [26–28, 30, 25]. It must be further studied if the efficient solutions they implement are compatible with our objective to support runtime assertion checking for such a rich specification language as E-ACSL. On the other hand, the ambitious objective to perform runtime assertion checking for C code completely specified in E-ACSL and directly compatible with integrated FRAMA-C tools for proof of programs (where manual analysis of proof failures can be even more costly) could justify a higher overhead.

Optimizations. Our proposal to record block metadata using Patricia tries is related to Jones's and Kelly's work [16] that proposed to use Splay trees for this purpose. Splay trees were also used in several recent tools related to memory safety [24, 29]. To the best of our knowledge, Patricia tries have never been used in this context. Static analysis based techniques to reduce memory monitoring have been used in earlier works, for instance, in [27, 28, 24]. Similarly, our dataflow analysis described in Sec. 6 performs an overapproximation of the necessary memory monitoring and successfully removes many irrelevant records and queries. We intend to further improve its precision in our future work.

9 Conclusion and future work

We have presented our solution of memory monitoring for runtime assertion checking with FRAMA-C. It can be applied on C code annotated in E-ACSL, an executable specification language offering among other features various memory-related constructs on validity and initialization of memory locations. The advantages of this solution include a very expressive specification formalism, a deep integration into the FRAMA-C platform and various possibilities of collaboration with other analyzers [7]. Thus, runtime assertion checking can benefit from annotations automatically generated by other plug-ins (e.g. VALUE or RTE), help to understand proof failures (e.g. during program proof with JESSIE or WP plug-ins), skip runtime checking of properties already established by other plug-ins and contribute to consolidated statuses of annotations in FRAMA-C [5].

We have described the global architecture, instrumentation with E-ACSL2C and particular aspects related to efficient storage of block metadata, efficient updates and lookups in the store and static analysis based optimization of monitored variables, as well as our initial experimental results. In particular, runtime assertion checking has indeed found errors in 100% of non-equivalent mutants for several simple C programs with complete E-ACSL contracts.

One future work direction is extending the support of E-ACSL language by E-ACSL2C, in particular, for temporal memory safety and advanced memory-related constructs like `\assigns`, `\freeable` or `\separated` [10]. Even if our main objective is different from many other works focused on memory safety, we would like to better evaluate our solution with respect to the state-of-the-art tools on commonly used benchmarks. Since we check only specified properties at runtime, that will require to write or automatically generate E-ACSL annotations related to memory safety. While runtime assertion checking for such a rich specification language as E-ACSL will likely have a greater overhead compared to these tools (that do not need to monitor function contracts and variable initialization, or treat specific memory-related E-ACSL constructs), some of the implementation solutions they used can still be applicable in our context. Future work also includes further optimizations to minimize the calls to the monitoring library (e.g. redundant checks or irrelevant monitoring).

References

1. Sullivan, M., Chillarege, R.: Software defects and their impact on system availability: A study of field failures in operating systems. In: the 1991 International Symposium on Fault-Tolerant Computing (FTCS 1991), IEEE Computer Society (1991) 2–9
2. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 25–37
3. Turing, A.: Checking a large routine. In: the Conference on High Speed Automatic Calculating Machines. (1949) 67–69
4. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5) (2009) 293–303
5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012). Volume 7504 of LNCS., Springer (2012) 233–247
6. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: the 28th Annual ACM Symposium on Applied Computing (SAC 2013), ACM (2013) 1230–1235
7. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: the 4th International Conference on Runtime Verification (RV 2013). LNCS, Springer (2013) To appear.
8. Szpankowski, W.: Patricia tries again revisited. J. ACM **37**(4) (October 1990) 691–711
9. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. (January 2012) URL: http://frama-c.com/download/ e-acsl/e-acsl.pdf.
10. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.6. (April 2013) URL: http://frama-c.com/acsl.html.
11. Baudin, P., Pacalet, A., Raguideau, J., Schoen, D., Williams, N.: CAVEAT: a tool for software validation. In: the 2002 International Conference on Dependable Systems and Networks (DSN 2002), IEEE Computer Society (2002) 537

12. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: the 19th Int. Conference on Computer Aided Verification (CAV 2007). Volume 4590 of LNCS., Springer (2007) 173–177
13. Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language, Iowa State Univ. (2003) URL: http://cs.iastate.edu/~leavens/JML/Relatedpapers/index.html.
14. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc. (1988)
15. ISO/IEC 9899:1999: Programming languages – C
16. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in c programs. In: the Third International Workshop on Automatic Debugging (AADEBUG 1997). (1997) 13–26
17. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: the 3rd International Conference on Virtual Execution Environments (VEE 2007), ACM (2007) 65–74
18. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: the 4th Int. Workshop on Automation of Software Test (AST 2009), IEEE Computer Society (2009) 70–78
19. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accomodates both runtime assertion checking and formal verification. In: the First International Symposium on Formal Methods for Components and Objects (FMCO 2002). Volume 2852 of LNCS., Springer (2002) 262–284
20. Comar, C., Kanig, J., Moy, Y.: Integrating formal program verification with testing. In: the Embedded Real-Time Software and Systems Congress (ERTS 2012). (2012)
21. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: In the 11th Annual Network and Distributed System Security Symposium (NDSS 2004). (2004) 159–169
22. Xu, W., DuVarney, D.C., Sekar, R.: An efficient and backwards-compatible transformation to ensure memory safety of C programs. In: the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004), ACM (2004) 117–126
23. Dhurjati, D., Adve, V.S.: Backwards-compatible array bounds checking for C with very low overhead. In: the 28th International Conference on Software Engineering (ICSE 2006). (2006) 162–171
24. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: the 18th USENIX Security Symposium (USENIX 2009), USENIX Association (2009) 51–66
25. Simpson, M.S., Barua, R.: MemSafe: ensuring the spatial and temporal memory safety of C at runtime. Softw., Pract. Exper. **43**(1) (2013) 93–128
26. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI 1994), ACM (1994) 290–301
27. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. **27**(3) (2005) 477–526
28. Oiwa, Y.: Implementation of the memory-safe full ANSI-C compiler. In: the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), ACM (2009) 259–269
29. Yuan, J., Johnson, R.: CAWDOR: compiler assisted worm defense. In: the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), IEEE Computer Society (2012) 54–63

30. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: the 2012 USENIX Annual Technical Conference (USENIX ATC 2012), USENIX Association (2012) 309–318