Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C

Arvid Jakobsson, Nikolai Kosmatov, Julien Signoles

CEA, LIST, Software Reliability and Security Laboratory, PC 174, 91191 Gif-sur-Yvette France

Abstract

One classical approach to ensuring memory safety of C programs is based on storing block metadata in a tree-like datastructure. However it becomes relatively slow when the number of memory locations in the tree becomes high. Another solution, based on shadow memory, allows very fast constant-time access to metadata and led to development of several highly optimized tools for the detection of memory safety errors. However, this solution appears to be insufficient for evaluation of complex memory-related properties of an expressive specification language.

In this work, we address memory monitoring in the context of runtime assertion checking of C programs annotated in E-ACSL, an expressive specification language offered by the FRAMA-C framework for the analysis of C code. We present an original combination of a tree-based and a shadow-memory-based techniques that reconciles the efficiency of shadow memory and the higher expressiveness of annotations that can be evaluated using a tree of metadata. Shadow memory with its instant access to stored metadata is used whenever small shadow metadata suffices to evaluate required annotations, while richer metadata stored in a compact prefix tree (Patricia trie) is used for evaluation of more complex memory annotations supported by E-ACSL. We also present a preliminary static analysis step that determines which variables should be monitored (and in which way) in order to be able to evaluate annotations present in the program.

The combined monitoring technique and the pre-analysis step have been implemented in the runtime assertion checking tool for E-ACSL. Our initial experiments confirm that the proposed hybrid approach leads to a significant speedup with respect to an earlier implementation based on a Patricia trie alone without any loss of precision, while the proposed static analysis reduces the monitoring of irrelevant variables and further improves the performances of the instrumented code.

Keywords: runtime assertion checking, memory monitoring, specification language, executable specification, Frama-C toolset

Email addresses: arvid.jakobsson@gmail.com (Arvid Jakobsson), nikolai.kosmatov@cea.fr (Nikolai Kosmatov), julien.signoles@cea.fr (Julien Signoles)

1. Introduction

Over the past few decades, memory safety of C programs has been addressed in numerous research efforts and tools. Many tools for dynamic verification answer questions regarding the memory of programs: how much memory is used, is memory correctly accessed, allocated and deallocated, etc. Such tools address memory-related errors, including invalid pointers, out-of-bounds memory accesses, uninitialized variables and memory leaks, that are very common. A study for IBM MVS software [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 critical system software and one of the most poorly equipped with adequate protection mechanisms. The C developer remains 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* has become a widely-used programming practice [2]. Runtime checking of memory-related properties can be realized using systematic monitoring of memory operations. However, to do so efficiently is difficult, because of the large number of memory accesses of a normal program. An efficient memory monitoring for C programs is the purpose of the present work.

This paper addresses the memory monitoring of C programs for runtime assertion checking in FRAMA-C [3], a platform for the analysis of C code. FRAMA-C offers 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 [4]. In order to support memory-related E-ACSL 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. E-ACSL2C comes with a runtime memory monitoring library for recording and retrieving necessary information (*memory block metadata*) on the state of the program's memory locations. During the translation of an original C code annotated with E-ACSL specification into a new C code, E-ACSL2C instruments the original source code by inserting necessary calls to the library. It realizes 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 current version of the library [5] records memory block metadata in a compact prefix tree (Patricia trie) [6], that appeared to be very efficient compared to other datastructures constructed on-demand. While the current metadata storage was subject to a careful choice of datastructures and optimizations [5], it remains one of the bottlenecks in terms of performance for programs instrumented by E-ACSL2C, which can be subject to a slowdown of more than 100x when the number of memory locations in the tree becomes high. Lookup operations in the Patricia trie still imply traversing the tree from the root to the node which contains the metadata needed, and thus several memory accesses.

Recent advanced tools for the detection of memory safety issues used an alternative approach, based on a statically allocated fixed array for metadata that allows a fast offset-based access [7, 8]. Such an array is called a *shadow memory*, since each address of the user-memory is shadowed by an element of this array. This approach assumes that a sufficiently long array for shadow memory can be allocated in the program memory, that is possible for the great majority of modern programs (except for architectures with very little memory available, or for programs using dispersed fixed addresses). In the context of runtime assertion checking of E-ACSL annotations, this approach alone would not be sufficient to store all metadata necessary to support various memory-related predicates offered by E-ACSL. Indeed, it can address properties on validity and initialization of memory locations, but it stores no information about the base address or the size of a given memory block, which is required to treat some memory-related E-ACSL predicates.

Goals. The first objective of this paper is to study how the existing tree-based memory monitoring solution can be improved using the shadow memory approach. We present an original combination of a tree-based and a shadow-memory-based techniques that reconciles the efficiency of shadow memory and the higher expressiveness of annotations that can be evaluated using a tree of metadata. Rather than providing detailed (but more difficult to follow) algorithms, we give comprehensive design principles of the combined technique. One current limitation of the technique is related to the detection of some subtle temporal errors[1] that are not yet fully supported. As usually in instrumentation-based techniques, we also assume that the complete source code of the target program is available.

Exhaustive memory monitoring of all program variables can be costly and is not necessary when only some of the variables occur in memory-related annotations. Our second objective is to describe another optimization that reduces irrelevant monitoring with respect to the user-defined verification objectives. We present a preliminary static analysis step (also called *pre-analysis* below) that computes an over-approximation of the set of memory locations whose monitoring is required to evaluate the given annotations. Moreover, the same pre-analysis step also determines which program variables should be monitored by the tree-based technique. We implement the combined monitoring technique and the pre-analysis step in the runtime assertion checking tool E-ACSL2C of FRAMA-C [3] and evaluate them on several experiments.

Contributions. The contributions of this paper include

- a classification of memory-related predicates of E-ACSL with regard to their monitoring level as byte-level and block-level,
- design and implementation for E-ACSL2C of a shadow memory storage of block metadata for byte-level annotations,

[1] For instance, in a progam fragment `p=malloc(N); q=p; free(p); p=malloc(N);`, pointer `q` that becomes dangling after the memory deallocation can be erroneously reported as valid again after the second allocation if it happens to allocate a new block at the same location.

- rough complexity evaluation for the Patricia trie and the shadow memory store,
- an original hybrid memory monitoring solution combining both kinds of storage,
- a rigorous semi-formal presentation of the pre-analysis step,
- implementation of the combined monitoring and pre-analysis for E-ACSL2C,
- evaluation of the proposed optimizations on several examples.

The present paper is an extended version of an earlier conference paper [9], completed in particular by a description of the pre-analysis (Sec. 5), a more detailed presentation of the E-ACSL specification language (Sec. 2), as well as additional experiments and analysis of results (Sec. 6).

Outline. The paper is organized as follows. Sec. 2 presents the E-ACSL specification language, while Sec. 3 describes the translation of the annotations into instrumented C code with E-ACSL2C. Sec. 4 introduces the monitoring level of memory-related predicates and describes the tree-based, the shadow-memory-based, and the hybrid monitoring solutions. Sec. 5 presents the pre-analysis step. These solutions are evaluated and compared in Sec. 6. Finally, Sec. 7 presents some related work, and Sec. 8 concludes the paper.

2. The E-ACSL Specification Language

Overview of E-ACSL. This section presents E-ACSL [4, 10], an executable specification language designed to support runtime assertion checking[2] in FRAMA-C.

FRAMA-C [3] is a framework dedicated to the analysis of C programs that offers various analyzers, such as abstract interpretation based plugin VALUE for value analysis, dependency analysis, program slicing, JESSIE and WP plugins for proof of programs, etc. ACSL [11] is a behavioral specification language shared by all FRAMA-C analyzers. It is inspired by JML [12, 13] and takes the best of the specification languages of earlier tools CAVEAT [14] and CADUCEUS [15].

ACSL is sufficiently rich to express most functional properties of C programs. It has already been used in many projects, including large-scale industrial ones [3]. 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 [16] may be associated to each function in order to specify its pre- and postconditions. The contract can be split into several named guarded behaviors. ACSL annotations also include assertions, loop invariants and loop variants, definitions of (inductive) predicates, axiomatics, lemmas, logic functions, data invariants and ghost code.

To illustrate ACSL, let us consider the C function `allZeros` of Fig. 1 annotated in ACSL. It checks whether all elements of given array `t` of size `n` are equal to 0 and returns a nonzero value in that case, and 0 otherwise. The function contract includes a

[2] *Runtime annotation checking* would be here a more suitable term since various kinds of annotations are supported.

```
1  /*@ requires n >= 0 && \valid( t + (0 .. n-1) );
2    @ assigns \nothing;
3    @ ensures \result <==> ( \forall integer i ; 0 <= i < n ==> t [ i ] == 0 );
4    @*/
5  int allZeros( int t[], int n ){
6    int k;
7    /*@ loop invariant \forall integer i ; 0 <= i < k ==> t [ i ] == 0;
8      @ loop invariant 0 <= k <= n;
9      @ loop assigns k;
10     @ loop variant n-k;
11     @*/
12   for( k = 0; k < n; k++ ){
13     if( t[k] != 0 )
14       return 0;
15   }
16   //@ assert \forall integer i ; 0 <= i < n ==> t [ i ] == 0;
17   return 1;
18 }
```

Figure 1: Function `allZeros` annotated in ACSL, returning a nonzero value if and only if all elements in the given array `t` of `n` integers are zeros

precondition (line 1) saying that `t` should be a valid array with $\texttt{n} \geq 0$ allocated elements (that is, these elements can be safely read and written). The postcondition at line 3 states that the returned value is nonzero if and only if all elements of `t` are zero. The frame rule (line 2) ensures that the function does not modify any non-local variable. The loop annotations specify the loop invariant (lines 7–8), loop variant (giving an upper bound on the number of remaining loop iterations, line 10) and the variables modified in the loop (line 9). The FRAMA-C/WP plugin can automatically prove that this function is correct with respect to the given specification.

E-ACSL [4, 10] is a large subset of ACSL that preserves ACSL semantics in the following sense. Every valid (resp. invalid) predicate in E-ACSL is valid (resp. invalid) in ACSL. Conversely, every valid (resp. invalid) predicate in ACSL is either valid (resp. invalid) or undefined in E-ACSL. *Undefined* predicates are those which would lead, according to the C standard [17], to an undefined runtime behavior when executed (e.g. predicates containing a division by zero or derefencing an invalid pointer). Indeed, the E-ACSL language is *executable*: its annotations can be translated into C monitors and executed at runtime. This makes it suitable for runtime assertion checking.

The requirement of being executable brings some natural limitations on ACSL annotations that can be supported in E-ACSL. E-ACSL syntactically limits quantifications to range over finite domains of integers in order to be computable. Loop invariants in E-ACSL lose their inductive nature: a loop invariant in E-ACSL is equivalent to two assertions: the first one before entering the loop and the second one at the end of each iteration of the loop body. In E-ACSL there are no lemmas (which usually express non-executable mathematical properties) nor axiomatics (non-executable by nature[3]). There is also no way to express termination properties of a loop or a recursive function, since

[3]It would be actually possible to generate executable code for some axiomatics, in particular those corresponding to (simple) inductive schemes. However, it is unclear what precise subset of ACSL axiomatics could be reasonably supported in E-ACSL. This study is left as future work.

E-ACSL keyword	Its semantics	Monitoring level
`\base_addr(p)`	base address of the block containing pointer `p`	Block
`\block_length(p)`	size (in bytes) of the block containing pointer `p`	Block
`\offset(p)`	offset (in bytes) of `p` in its block	Block
`\valid_read(p)`	is true iff reading `*p` is safe	byte
`\valid(p)`	is true iff reading and writing `*p` is safe	byte
`\initialized(p)`	is true iff `*p` has been initialized	byte

Figure 2: Memory-related E-ACSL constructs currently supported by E-ACSL2C. In the last three constructs, `p` must be a non-void pointer.

```
1  int main(){
2    int arr[10]={3,1,4,1,5,9,2,6,5,3}, subarr[3]={4,1,5}, *result;
3    unsigned len=10, sublen=3;
4    //@ assert \forall int i; 0 <= i < len ==> \valid(arr+i);
5    //@ assert \forall int j; 0 <= j < sublen ==> \valid(subarr+j);
6    // search an occurrence of the list subarr in the list arr
7    unsigned i, j, found = 0;
8    for(i = 0; i <= len-sublen; i++){
9      found = 1;
10     for(j = 0; found && j < sublen; j++)
11       if(arr[i+j] != subarr[j])
12         found = 0;
13     if(found)
14       break;
15   }
16   if(found){
17     result = arr+i;       // found, result points to the occurrence
18   // check the result
19   //@ assert \base_addr(arr) == \base_addr(result);
20   //@ assert \offset(arr)<=\offset(result)<=\offset(arr)+(len-sublen)*sizeof(int);
21   //@ assert \forall int j; 0 <= j < sublen ==> result[j] == subarr[j];
22   } else
23     result = (void*)0;    // not found, NULL
24 }
```

Figure 3: Program `findSubarray` that looks for an occurrence of array `subarr` with `sublen` integers as a subarray in array `arr` with `len` integers, and assigns to `result` the pointer to such an occurrence if found

detecting non-termination at runtime in finite time is not possible. All other features of ACSL are fully supported in E-ACSL, including mathematical integers, predicates and functions over C pointers. We refer the reader to [10] for a systematic description of E-ACSL constructs.

The first two columns of Fig. 2 present some memory-related annotations supported by E-ACSL. (The third column will be explained in Section 4.) 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. The offset of a pointer inside its block is computed with respect to the base address.

Example. Fig. 3 contains a toy example illustrating memory-related predicates of E-ACSL. The code at lines 6–17, 22–23 checks if the array of integers `arr` contains another array `subarr` as a subarray. If an occurrence of the subarray is found, `result`

will point to the first element of such an occurrence. Otherwise, it will be null. For the given values of arrays and their lengths (lines 2–3), the sublist `subarr` will be found starting from index 2 in `arr`. This simplified example is inspired by the `strstr` standard library function that looks for a substring in a given string. The assertion at line 4 of Fig. 3 states that the cells at indices `0..len-1` of array `arr` must be valid, while that of line 5 checks the validity of elements in `subarr`. The assertions at lines 19–21 ensure that `result` points to an occurrence of `subarr` in `arr`. We write them in the most general form for any values of arrays and covering the case where the arrays can be part of bigger blocks, so `arr` is not necessarily a base address itself like in this toy example. We check that `arr` and `result` belong to the same block (line 19), that `result` points to an element inside the appropriate part of `arr` (line 20) and that this element starts indeed an occurrence of the subarray `subarr` (line 21).

3. Runtime Assertion Checking of E-ACSL Annotations

Instrumentation of C code with E-ACSL2C. 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.) relies on a non-invasive instrumentation[4] of C code by E-ACSL2C as described in [4]. However, runtime assertion checking of E-ACSL specifications involving memory-related constructs of Fig. 2 requires particular care.

In order to evaluate memory-related E-ACSL annotations (cf. Fig. 2), we record metadata on validity, initialization, size, etc. of memory locations during program execution in a dedicated data store, that we call *the store.* The instrumented code relies on 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 (allocation, deallocation, initialization) in the store.

For instance, assuming **`sizeof`**`(`**`int`**`)` is 8 bytes, the predicate **`\valid`**`(arr+i)` at line 4 of Fig. 3 is translated into a call to a library primitive `__valid(arr+i, 8)` that is supposed to query the store and to determine if the pointer `arr+i` is valid. It can be determined since E-ACSL2C records the automatic allocation of an array of 10 integers by inserting another library call `__store_block(arr, 8*10)` after the allocation at line 2. It also deregisters this allocation from the store by inserting a library call `__delete_block(arr)` at the end of the scope. Similarly, all other memory-related operations (static, dynamic and automatic allocations, deallocations and initializations) are instrumented as well, as presented in more detail in the example of Sec. 5 and in [5].

[4] A formal description of a translation similar to E-ACSL2C in a different context (test generation instead of monitoring) can be found in [18].

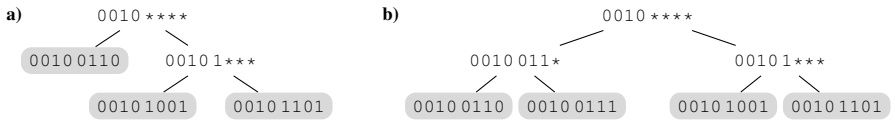


Figure 4: Example of a compact prefix tree (Patricia trie) **a)** before, and **b)** after inserting `0010 0111`

Two ways to optimize the instrumented code. In this instrumentation-based approach, frequent calls to library primitives can significantly decrease the performance of the instrumented program. One way to reduce this slow-down is to reduce the number of library calls, that is, to restrict the monitoring to memory locations that are necessary to evaluate the existing memory-related annotations, and avoid the monitoring of irrelevant ones. E-ACSL2C offers a pre-analysis step that performs a backward data-flow analysis computing an over-approximated set of memory locations that are sufficient to monitor in order to verify memory-related annotations. For example, to evaluate the annotations of the program in Fig. 3 it would be sufficient to monitor the locations pointed by `arr`, `subarr` and `&result` (the last one being required e.g. to check if `result` is valid before dereferencing `result` at line 21). The data-flow analysis will indeed exclude from monitoring the locations at addresses `&len`, `&sublen`, `&i` and `&j`. If the assertion of line 5 were removed from the program, `subarr` would be excluded from monitoring as well, and thus only `arr` and `&result` would be monitored. The pre-analysis step is described in detail in Section 5.

Another way to improve the performance of instrumented code is providing an efficient design of the store, since the efficiency of the instrumented code strongly depends on the speed of the library. This is the purpose of the following section.

4. Hybrid Memory Monitoring

This section presents our hybrid memory monitoring solution. First, we introduce the notions of *byte-level* and *block-level* monitoring. Then we present the tree-based storage and the shadow-memory-based storage approaches, give some complexity evaluation insights and descibe the proposed combination. Finally, we illustrate our solution on an example.

Byte-Level and Block-Level Monitoring. Memory-related annotations of E-ACSL presented in Fig. 2 can be classified into two groups. On the one hand, the predicates related to validity and initialization of the memory location referred to by `p` do not require any information of the relative position (offset) of `p` in its block, or the size or base address of this block. Local (validity or initialization) information for the specific bytes composing `*p` is sufficient to evaluate these predicates. We say that these predicates require *byte-level* monitoring (as indicated in the third column of Fig. 2). On the other hand, local information does not suffice for the first three predicates whose evaluation requires global block characteristics: base address and size. We say that such predicates require *block-level* monitoring.

For instance, the assertions at lines 4–5 in Fig. 3 include only byte-level predicates, while the assertions at lines 19–20 include block-level constructs. Since the assertion at

line 5 is the only assertion to be evaluated for `subarr` (or its aliases), byte-level monitoring of the array `subarr` would be sufficient. The array `arr` requires block-level monitoring because of the assertions at lines 19–20. The required level of monitoring of program variables can be determined by the pre-analysis step as we show in Section 5.

Let us now present a tree-based and a shadow-memory-based storage of user memory metadata, as well as the hybrid memory monitoring solution we propose to support memory-related annotations of E-ACSL.

Tree-Based Storage of Metadata. A previous work [5] proposed a memory monitoring solution for E-ACSL based on a compact prefix tree (Patricia trie) [6] for storing block metadata. The keys of tree nodes are base addresses (that is, 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 which contains the greatest common prefix of base addresses stored in its successors. Fig. 4a illustrates 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 one of undefined bits following the greatest common prefix. Fig. 4b 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`$\star$. Conversely, removing `0010 0111` from the trie of Fig. 4b would give that of Fig. 4a.

Although the Patricia trie came out on top out of the different on-demand-constructed datastructures which were evaluated during the design of the store [5], it has still a significant cost attached to updating the store while the verified program is running, as well as the cost of querying the store to retrieve the metadata. The trie is a tree structure, and each level of the tree explored during lookup incurs at least one memory read. This means that programs using a big number of variables will have a much longer lookup time for metadata than a program using a smaller set of variables.

An advantage of this datastructure is a potentially big amount of block metadata that can be attached to a node for all required information, including base address, validity, block size, initialized bytes, dynamic (freeable) or non-dynamic origin, writable or read-only, etc. Therefore, this solution can support both byte-level and block-level monitoring.

Shadow-Memory-Based Storage of Metadata. An efficient alternative to a tree-based storage is to use a linear structure with an offset-based access to metadata. Such a store is called a *shadow memory* [7, 8], since each address of the user-memory is shadowed by an element of this structure. A shadow memory is a large array, such that an address of the user-memory can be associated with an element of this array. The mapping $\mathrm{Sh}(p)$ of a user memory address p to the address in the shadow memory where the corresponding metadata is stored is basically a linear mapping:

$$\mathrm{Sh}(p) = \mathrm{Sh\_Base} + p \cdot \mathrm{Scale}$$

obtained by a simple offset $\mathrm{Sh\_Base}$ with a scale (if more, or less, than one byte of shadow memory should be associated to each byte of the user memory). It can be

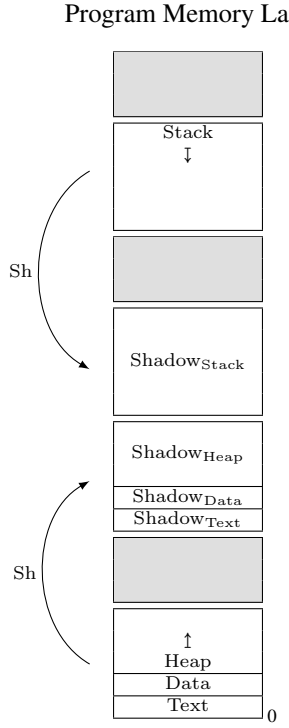


Figure 5: Metadata storage based on shadow memory

noted that if Scale is a power of 2, then the multiplication can be performed by a rapid bitshift.

The design of the new shadow-memory-based store for memory monitoring with E-ACSL is illustrated in Fig. 5. In a standard memory layout of the program execution under Linux, most user-created memory allocations tend to cluster in the top and bottom of the addressable space. The stack sits in the top and grows downwards, while the text and data segments are fixed in the bottom, with dynamically allocated memory on top of them growing upwards. The areas in the middle of the memory are rarely used except for the mapping of dynamically linked libraries, which are actually not fully supported by the store. Since the addressable memory of a program is far bigger than what most programs really use during execution, a linear structure that covers a portion of the upper area and the lower area, but ignores the middle area, can shadow the memory actually used by most programs. We allocate such a block for the shadow memory whose size was set to the largest continuous mapping returned by `mmap`. It is divided into two parts: the higher one, $\mathrm{Shadow}_{\mathrm{Stack}}$, to store metadata of the stack, and the lower one, containing $\mathrm{Shadow}_{\mathrm{Text}}$, $\mathrm{Shadow}_{\mathrm{Data}}$ and $\mathrm{Shadow}_{\mathrm{Heap}}$, for the metadata of the corresponding three areas. Gray areas in Fig. 5 show memory zones that are unaddressable and/or not represented by the shadow memory store. An access to the metadata for an address p is preceded by an address check, that detects if p does not

belong to the address intervals modeled by the shadow memory.

Separate bits of the shadow of a byte are used to represent its validity and initialization status. The bytes of each newly allocated block are first marked, one after another, as valid and uninitialized. When some bytes are assigned a value, each of these bytes is marked as initialized. Finally, when the block is deallocated, each of its bytes is marked as invalid in the shadow memory. In this way, validity and initialization of memory locations can be efficiently updated and evaluated during the execution of the instrumented code.

If the metadata of a block cannot be found to set validity or initialization, then it means that the block is not in the area covered by the shadow memory. In this case the memory monitoring library stops the execution of the program with a warning.

Thus, a limitation of shadow memory techniques is the requirement to allocate a long continuous memory block to store the metadata. Indeed, to be efficient, the shadow memory is not constructed on-demand like the Patricia trie, it must be allocated from the beginning until the end of the program. For example, if one byte of metadata is stored for each byte of memory used by the analyzed program, then half of the memory must be allocated to the memory monitoring library. This can be infeasible for some programs with strong memory constraints (e.g. using dispersed fixed addresses). Since most programs do not have such constraints and do not use most of the available virtual memory, it is often possible to allocate a sufficiently long continuous memory block for shadow memory.

In the context of runtime assertion checking for an expressive specification language like E-ACSL, another important limitation of the shadow memory is that it is only suitable for byte-level monitoring since metadata is associated to each unique byte. Associating all block metadata to each byte would need even more shadow memory and appeared to be too costly in terms of memory for our purpose.

Complexity Evaluation Insights. To better understand the strong and the weak points of both techniques before describing their combination, let us give rough upper bounds on complexity of additional memory monitoring code in both cases. For simplicity, we do not consider cache related issues (that could also impact final performance), and suppose we monitor and evaluate only validity information without pre-analysis on a 64-bit architecture. Let A be the number of (dynamic, static and automatic) allocations of memory blocks, L be the maximal size of these blocks, D be the number of deallocations, and V the number of evaluations of a **`\valid`** predicate for locations (again, of maximal length of L bytes). All numbers are counted over the whole program execution. Each of these operations requires a lookup to find the place of the corresponding (added, deleted or searched) node in the trie. Let H be the maximal height of the Patricia trie, so $H \leq 1 + 64$. Moreover, if the trie is balanced (that is not at all ensured for the Patricia trie) and contains at most N blocks stored in its leaves, then $H \leq 1 + \log N$, otherwise, in an unbalanced tree, we have $H \leq 1 + N$. Let m_a and m_d be the number of steps required by node allocation and deallocation for a node insertion or removal, while I_{Sh} denotes the number of steps for shadow memory initialization. The number of additional operations for monitoring allocations, deallocations and validity checks using a Patricia trie can be (very roughly) bounded by

$$\leq A \cdot (H + m_a) + D \cdot (H + m_d) + V \cdot H, \tag{1}$$

not depending on L, while the shadow memory store would lead to

$$\leq I_{Sh} + A \cdot L + D \cdot L + V \cdot L \tag{2}$$

additional steps, each of the block bytes being treated separately. These estimates illustrate that the Patricia trie can be expected to be less efficient when H becomes high and L remains low (i.e. many small memory blocks are monitored), and more efficient when the trie remains small while high block sizes (or, for fast programs, shadow initialization I_{Sh}) penalize the shadow memory approach.

The Combined Monitoring Solution. The objective of the combined memory monitoring solution is to reconcile the benefits of both tree-based and shadow-memory-based storage within the same library. As we have explained above, the Patricia trie is often slower but supports most expressive memory-related annotations. On the other hand, the shadow-memory technique is usually much faster, but is not suitable for block-level monitoring. Let us present the main principles of the proposed combined memory monitoring solution.

Metadata for a newly allocated block should be recorded using the following principles:

P1 Metadata of all memory locations that require block-level monitoring should be stored in the Patricia trie store.

P2 For memory locations that require byte-level monitoring, their metadata are typically stored in the shadow memory store.

The required level of monitoring for program memory locations is statically computed by the pre-analysis step presented in Sec. 5.

In addition, there are some specific cases and optimizations where the Patricia trie store is used rather than shadow memory after a runtime check, according to the size and range of addresses of a particular block.

P3 If a block does not completely belong to the interval of addresses supported by the shadow memory, its metadata should be stored in the Patricia trie store.

While this situation has never occurred in our benchmarks, Principle **P3** addresses the restriction on the supported address interval of the shadow memory store.

If a block has a relatively big size, its allocation, deallocation or query would require to access a long interval of metadata bytes in the shadow memory, that may be longer than treating the block in the Patricia trie (cf. estimates (1) and (2) above). Therefore we use the following additional optimization:

P4 For blocks longer than a (parameterizable) constant length C, the metadata should be stored in the Patricia trie store.

The evaluation of a memory-related annotation is basically realized in the following way.

P5 The evaluation of block-level predicates should always query the Patricia trie.

Code	Monitoring Action	Monit. Level	Principle
1 `struct BigStruct a[2];`	record (`&a[0]`,64) in PT	byte	**P4**
2 `struct BigStruct b;`	record (`&b`,32) in PT	Block	**P1**
3 `//@ assert \block_length(&b)==1;`	query `&b` in PT	Block	**P5**
4 `char c;`	record (`&c`,1) in Sh	byte	**P2**
5 `//@ assert \valid_read(&c);`	query `&c` in Sh	byte	**P6**
6 `struct BigStruct *p=&a[1];`	record (`&p`,8) in Sh	byte	**P2**
7 `//@ assert \valid(p);`	query `p` in Sh, PT	byte	**P6**

Figure 6: Example illustrating the combined memory store with $C = 32$ bytes, assuming a 64-bit architecture and `sizeof(struct BigStruct)`=32. The Monitoring Action indicates the block base address and size being recorded, or the address being looked up, in one (or, when necessary, both) of the stores.

P6 For predicates that require byte-level monitoring, the evaluation first tries to find the required information in the shadow memory store. If the block is unknown in the shadow memory store, it queries the Patricia trie store.

It means that the evaluation of a byte-level memory-related annotation starts by interrogating the shadow memory store (with almost instant access) and queries the slower Patricia trie store afterwards only when necessary (when the first byte of the block is not known as valid in the shadow memory).

The deallocation of a memory block is basically realized in the following way.

P7 The deallocation of an existing block first tries to remove the block metadata from the shadow memory store. If the block is unknown in the shadow memory store, it queries and tries to remove it from the Patricia trie store.

Here again, the library starts by interrogating the shadow memory store and queries the slower Patricia trie store only if necessary.

Example. Fig. 6 illustrates these principles on a simple example, where we assume a 64-bit architecture, the size of an element of type `struct BigStruct` equal to 32 bytes, and the constant C set to 32 bytes. The figure shows how the combined memory store is updated and queried by the instrumented code during the monitoring of the code in the leftmost column of the table. For each line of code, the second column describes the monitoring action, that is either recording a couple (base address, size) for a new block, or querying the store for an address. (For simplicity, the complete instrumented code is omitted, and the actions to monitor initialization of variables are not shown either.) It also indicates the store being used: the shadow memory store (Sh), the Patricia trie store (PT) or, when necessary, both. The third column gives the monitoring level of memory-related annotations or introduced variables (we suppose that the monitoring level for program variables has been computed by the pre-analysis step described in Sec. 5). The last column contains the principle used to select which store should used by the monitoring operation.

The code in the example allocates four variables: a, b, c and p. The pre-analysis phase will deduce that block-level monitoring is needed for (the block with) base address `&b` because of its presence in the block-level predicate on line 3, but that byte-level monitoring suffices for base addresses `a`, `&c` and `&p`.

On the first line, an array `a` of $32 \times 2 = 64$ bytes is allocated. Even if `a` requires only byte-level monitoring, since its size is greater than $C = 32$, it is recorded in the Patricia trie store according to **P4**. The second line allocates the variable `b` of 32 bytes for which block-level monitoring is required, so `&b` is recorded in the Patricia trie store as well, following **P1**. On line 3, a **`\block_length`** predicate is evaluated. Since this is a block-level predicate, by **P5** the Patricia trie store is queried.

Line 4 allocates a single char `c` whose size (1 byte) is not greater than C. In contrast to `b`, no block-level metadata is needed for this variable according to the pre-analysis, thus the meta-data for base address `&c` is stored in the shadow memory, by **P2**. On line 5, a byte-level predicate queries the shadow memory store (where is finds the corresponding metadata and does not need to query the Patricia trie store afterwards, cf. **P6**).

A pointer `p` is created on line 6, pointing to the second element of array `a`. The meta-data for `&p` will be stored in the shadow memory by **P2** (since the pointer size is 8 bytes, Principle **P4** does not apply). Finally, on line 7, to evaluate the byte-level predicate **`\valid`**, the shadow memory will be queried first according to **P6**. Since the metadata for `a` is stored in the Patricia trie, already the first byte pointed by `p` is unknown to the shadow memory, so a second query is rapidly performed in the Patricia trie. The metadata of `a` is found and the validity of `p` can be established.

This simple example illustrates how the proposed combination of recording/query strategies supports expressive memory-related constructs of E-ACSL, and tries to avoid a byte-after-byte verification of validity for all bytes of big datastructures, as well as more time-consuming queries in the Patricia trie for small variables.

5. Pre-analysis

The previous section has introduced the hybrid memory store based on two different datastructures, a shadow memory store for byte-level and the Patricia trie store for block-level monitoring. This combined monitoring solution is based on 7 principles, **P1** to **P7**. They assume that, for each memory block, we know whether it requires block-level monitoring or whether byte-level monitoring is sufficient to evaluate memory-related annotations dealing with this block.

In this section, we introduce a preliminary static analysis step that can be used to compute the monitoring level for each memory block before instrumenting the program with E-ACSL2C. It performs an over-approximating backward dataflow analysis that allows E-ACSL2C to automatically decide whether a given memory block should be recorded in the Patricia trie or in the shadow memory store. The pre-analysis also computes whether it is *necessary* to monitor the block at all.

More precisely, the pre-analysis determines, for every block, whether it is necessary (i) to monitor its validity, (ii) to monitor its initialization, and (iii) to use block-level monitoring for this block. Using the pre-analysis eventually leads to an optimized instrumentation (both in terms of memory- and time-consumption, cf. Section 6). For instance, if some block is never concerned by an annotation, there is no need to monitor this block at all.

```
1  int main(void) {
2    int x, y, z, *p, *q;
3    q = (int*)malloc(10 * sizeof(int));
4    y = 1;
5    z = 2;
6    /*@ assert \initialized(&y); */
7    x = 0;
8    p = &x;
9    /*@ assert \valid(p); */
10   while (x < 10) {
11     *(q+x) = x;
12     x++;
13   }
14   /*@ assert \block_length(q) == 10 * \block_length(p); */
15   return 0;
16 }
```

Figure 7: Toy example illustrating the benefits of reducing the instrumentation.

Example. Consider for instance the toy example of Fig. 7. It performs a few pointer manipulations and memory checks through E-ACSL annotations.

Fig. 8 shows the result after a complete (non-optimized) instrumentation of this program with E-ACSL2C. It uses the primitives provided by the monitoring library. First, for each local variable, a memory block of corresponding size must be recorded in the store by calling the dedicated function `__store_block` (cf. lines 4–8 of Fig. 8). The instrumentation also replaces the call to `malloc` by a custom one (line 10 of Fig. 8) that records the newly allocated memory block in the store in addition to its allocation[5]. Then, for each assignment, the assignee is declared to be initialized through a call to `__full_init` (for all bytes of the block) or `__initialize` (for a given number of bytes, cf. line 27 of Fig. 8).

Thanks to these registrations in the store, the instrumented program is capable to evaluate the E-ACSL annotations. Before checking whether pointer `p` is valid (cf. lines 22–25 of Fig. 8), the instrumented program first ensures that `p` is initialized (since reading an uninitialized pointer `p` has an undefined behavior even if we just check its validity). Similarly, before evaluating the block sizes of `q` and `p` (cf. lines 32–36 of Fig. 8), the validity of `q` and `p` is checked, which, in turn, requires to check again that these pointers have been initialized. Otherwise, the evaluation of annotations in this example is straightforward. It relies on the monitoring library primitives that query the store to retrieve the desired information. Finally, the instrumented program removes the allocated variables from the store at the end of their scope by calling `__delete_block` (cf. lines 38–42 of Fig. 8).

Such a complete instrumentation leads to an exhaustive memory monitoring and can often be greatly improved by removing irrelevant monitoring operations. Indeed, in our example, no annotation is related to `z`, thus its monitoring is useless (cf. lines 6, 13 of Fig. 8). At a first glance, the situation seems to be the same for `x`. That is not true: since `p` is involved in some annotations and becomes an alias for `&x` at line 8 of Fig. 7, it is actually necessary to register base address `&x` in the store. However, since

[5]The block is recorded in the store only if its allocation does not fail.

```
1  int main(void) {
2    int x, y, z, *p, *q;
3    // allocation of locals
4    __store_block(& q,8);
5    __store_block(& p,8);
6    __store_block(& z,4);    // USELESS
7    __store_block(& y,4);
8    __store_block(& x,4);
9    __full_init(& q);        // initialization of q
10   q = (int *)__e_acsl_malloc(10 * sizeof(int));
11   __full_init(& y);        // initialization of y
12   y = 1;
13   __full_init(& z);        // initialization of z, USELESS
14   z = 2;
15   /*@ assert \initialized(&y); */
16   // check the above assertion
17   e_acsl_assert(__initialized(& y,sizeof(int)));
18   __full_init(& x);        // initialization of x, USELESS
19   x = 0;
20   __full_init(& p);        // initialization of p
21   p = & x;
22   /*@ assert \valid(p); */
23   // check the above assertion
24   e_acsl_assert(__initialized(& p,sizeof(int *));
25   e_acsl_assert(__valid(p,sizeof(int));
26   while (x < 10) {
27     __initialize(q+x,sizeof(int));  // initialization of *(q+x), USELESS
28     *(q+x) = x;
29     __full_init(& x);               // initialization of x, USELESS
30     x ++;
31   }
32   /*@ assert \block_length(q) == 10 * \block_length(p); */
33   // check the above assertion
34   e_acsl_assert(__initialized(& q,sizeof(int *) && __initialized(& p,sizeof(int *));
35   e_acsl_assert(__valid(q) && __valid(p));
36   e_acsl_assert(__block_length(q) == 10 * __block_length(p));
37   // clean the allocated memory
38   __delete_block(& q);
39   __delete_block(& p);
40   __delete_block(& z);     // USELESS
41   __delete_block(& y);
42   __delete_block(& x);
43   return 0;
44 }
```

Figure 8: Complete instrumentation with E-ACSL2C of the program of Fig. 7 for a 64-bit architecture.

the instrumented program does not check whether `x` is initialized, it is not necessary to monitor this specific information (through the call to `__full_init(&x)`, cf. lines 18, 29 of Fig. 8). Similarly, the instrumented program does not need to monitor the initialization of `*(q+x)` (cf. line 27 of Fig. 8). Thus all lines marked `USELESS` in Fig. 8 can be safely removed from the instrumented program.

Furthermore, coming back to the monitoring level of memory blocks, it is possible to statically deduce from this program that the blocks with base addresses `&x` and `q` must be registered in the Patricia trie store, while it is sufficient to register base addresses `&y`, `&p` and `&q` in the shadow memory store. Indeed, the instrumented program only queries the store for initialization of `y`, `p` and `q` (byte-level information) whereas it requires both byte-level and block-level information for base address `&x` (via its alias `p`), and block-level information only for base address `q`.

Formal Language. Introducing our static analysis formally for such a rich and complex language as C would not be not tractable in the present paper. Therefore, we limit this study to a smaller imperative language, defined in Fig. 9, that remains nevertheless representative to explain the key features of the pre-analysis.

This language contains pure (that is, side-effect-free) expressions with pointer accesses, addresses and offset shifts (denoted by ++ to avoid any confusion with addition of integers), as well as usual arithmetic operations (not detailed in Fig. 9). *Left values* (or *lvalues*) are either variables or dereferenced pointers. We use the term *address value* for any memory value of a pointer type (where *memory values* are defined in Fig. 9). The statements are assignments, dynamic allocations of a memory block (whose size is equal to the size of the given type κ multiplied by the given number of elements n), deallocations, conditionals, loops, sequences and blocks. Any statement can embed predicate-based assertions. They contain comparison operators, conjunctions, disjunctions, negations and the memory-related predicates introduced previously. The logical terms include pure expressions, the memory-related functions introduced above as well as arithmetic operations that may combine both[6].

In the rest of the section, we assume that variable names are unique (which can always be achieved by renaming). The typing system and operational semantics of this language are standard and can be omitted. We just indicate that the semantics is blocking [19, 20, 21]: if an error occurs when running the program, the execution stops immediately. We also suppose that each (logical) term is well defined, so its runtime evaluation does not raise any runtime error. For instance, no term leads to a division by zero or dereferencing an invalid pointer. This hypothesis might look too strong, but it is fully consistent with the E-ACSL semantics [10]. It is indeed enforced by E-ACSL2C, which automatically generates all necessary guards preventing such runtime errors during the evaluation of annotations [4] (cf. lines 32–36 of Fig. 8).

The formal definition of the pre-analysis relies on the notion of *address value base* of an address value a, denoted $\mathrm{base}(a)$, that is obtained from a by ignoring any offset shift.

Definition 1 (Address value base) base *is a function mapping an address value to an address value, inductively defined as follows:*

$$\begin{aligned}
\mathrm{base}(x) &= x \\
\mathrm{base}(\star a) &= \star\mathrm{base}(a) \\
\mathrm{base}(a \;{+}{+}\; n) &= \mathrm{base}(a) \\
\mathrm{base}(\&x) &= \&x \\
\mathrm{base}(\&(\star a)) &= \mathrm{base}(a).
\end{aligned}$$

Notice that this definition of address value base $\mathrm{base}(a)$ is different from the block base address defined e.g. by the E-ACSL construct `\base_addr(a)`. Indeed, the

[6] Adding integer constants, addition and multiplication in expressions and terms would be necessary to express all instructions and annotations of the example of Fig. 7. The presented analysis can be easily extended to these features.

left values	l ::=	x	variable
	|	$\star a$	pointer access
memory values	a ::=	l	left value
	|	a ++ n	pointer offset
	|	$\&l$	address
expressions	e ::=	a	memory value
	|	$\cdots$	pure expressions
statements	s ::=	;	skip
	|	l = e;	assignment
	|	l = `alloc`(κ, n);	allocation
	|	`free`(l);	deallocation
	|	`if` (e) `then` s; `else` s;	conditional
	|	`while` (e) s;	loop
	|	$s\,s$	sequence
	|	$\{s\}$	block
	|	`/*@ assert` p; `*/` s	assertion
predicates	p ::=	$t \equiv t \mid t \leq t$	comparators
	|	$p \wedge p \mid p \vee p \mid \neg p$	logic connectors
	|	`\valid`(a)	writing/reading $\star a$ is safe
	|	`\valid_read`(a)	reading $\star a$ is safe
	|	`\initialized`(a)	$\star a$ has been initialized
terms	t ::=	e	pure expressions
	|	`\base_addr`(a)	base address of a's memory block
	|	`\block_length`(a)	size of a's memory block
	|	`\offset`(a)	offset of a in its memory block
	|	$\cdots$	pure expressions combined with memory-related constructs
types	κ ::=	ι	integral type
	|	$\kappa\star$	pointer type

Figure 9: Formal syntax of the considered language.

definition of $\mathrm{base}(a)$ is based on the syntactical expression of a, while the base address is defined as the address of the first byte of the memory block effectively containing the current physical address a. For instance, if the pointer `p=a+4` points to the fifth element of an array `a` of 10 integers, then both **`\base_addr`**`(p+2)=a` and **`\base_addr`**`(a+6)=a`, but base`(p+2)=p` and base`(a+6)=a`.

The presented analysis also relies on the control flow graph of a program P. Our definition is directly adapted from Nielson *et al.* [22].

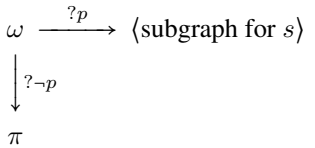
Definition 2 (Control flow graph [22]) *A* control flow graph (CFG) *of a program P is a sextuple $(\mathcal{V}, \mathcal{T}, \mathcal{E}, \pi, \mathcal{I}, \mathcal{F})$ such that:*

- *$\mathcal{V}$ is the set of vertices of the graph: for each statement s of P, two of these vertices are associated to s and represent its previous and next states. Each vertex $\omega \in \mathcal{V}$ corresponds to a unique program point in P (and we will sometimes refer to a CFG vertex ω as a program point ω).*
- *$\mathcal{T} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges.*
- *$\mathcal{E} : \mathcal{T} \to \mathcal{L}$ is a function mapping an edge to a label, where the set of labels $\mathcal{L}$ defined by:*

  $$\mathcal{L} = \{\texttt{skip},\ l = e,\ l = \texttt{alloc}(\kappa, n),\ \texttt{free}(l),\ ?e,\ ?p,\ \texttt{error}\}.$$

  The first label corresponds to the execution of the statement `;`, while the second, third and fourth ones respectively correspond to the assignment $l = e$, the allocation $l = \texttt{alloc}(\kappa, n)$ and the deallocation $\texttt{free}(l)$. Label $?e$ indicates that the program is branching and the edge is executed with the guard e being valid. Label $?p$ indicates that the edge corresponds to the execution of a valid predicate p, and the last label corresponds to a runtime error.
- *$\mathcal{I} \in \mathcal{V}$ (resp. $\mathcal{F} \in \mathcal{V}$) is the initial (resp. final) state of P, and $\pi \in \mathcal{V}$ is the error state.*

The construction of a control flow graph is straightforward (and omitted here) using the correspondence between labels and statements given in the definition. Details are provided in Nielson *et al.* [22]. We just draw the reader's attention to the statement `/*@ assert p; */` s : if ω is the state before the assertion, the subgraph for this statement is constructed as follows:



Dataflow Analysis. Our analysis is an over-approximating backward dataflow analysis parameterized by an alias analysis. We assume the existence of such an analysis.

Assumption 1 (Alias function) *For any given program P and a given program point ω in P, there is a function*[7] $\mathcal{A}^\omega$ *that maps an address value a to an over-approximation of the set S_a^ω of address values which are in alias with a at program point ω, including a itself (i.e. $a \in S_a^\omega \subseteq \mathcal{A}^\omega(a)$).*

Since the pre-analysis is only aimed at optimizing the instrumentation, the analysis speed is more important than its precision. Consequently, the alias analysis can be provided by Steensgaard's analysis [23], or the FRAMA-C Value analysis [3] if it has already been executed.

For a pointer p, in order to avoid any confusion between the left value p and its pointed value $\star p$ (which is also a left value), we are going to monitor the *address* $\&l$ of a left value l. Also, for the sake of performance, we are going to monitor a (dynamic) array *as a whole* through its base address rather than considering each of its single cells independently. Therefore, the analysis only needs to deal with address value bases, in which the offset shifts are removed, in other words, with address values a such that $a = \mathrm{base}(a)$.

The goal of the analysis is to determine for each address value base a that occurs in the program, with which observation purpose(s) a should be monitored. We consider three *observation purposes* denoted v, i, b, where v requires to monitor *validity*, i requires to monitor *initialization* and b requires *block-level* monitoring. The purpose v means that the block should be recorded in the store (and, thus, its validity can be queried). The purposes i and b are more specific, and therefore, each of them implies the presence of the purpose v as well. At every moment, the state σ of the analysis is a set of couples (a, k) where the presence of a couple (a, k) indicates that an address value base a should be monitored with the observation purpose $k \in \{v, i, b\}$.

Following Nielson *et al.* [22], we use an analysis framework that automatically computes the analysis state $\sigma \in \mathcal{S}$ at each program point by joining post-branch states and computing a fixpoint of loops. Our backward dataflow analysis is defined by a quadruple $(\mathcal{S}, \subseteq, \epsilon, s_0)$ as follows.

- $(\mathcal{S}, \subseteq)$ is a lattice of sets of couples (a, k), ordered by set inclusion, in which a is an address value base and $k \in \{v, i, b\}$ is an observation purpose.
- $s_0 \in \mathcal{S}$ is the initial state of the analysis, that is, the state at the final vertex of the control flow graph (as we define a *backward* dataflow analysis). Since no memory address needs to be monitored at this point, $s_0 = \varnothing$.
- $\epsilon_\lambda^\omega : \mathcal{S} \to \mathcal{S}$ is a set of monotonic transition functions over states, indexed by a graph edge label $\lambda \in \mathcal{L}$ and a particular program point $\omega \in \mathcal{V}$, defined in Fig. 10. Informally speaking, given an edge with origin vertex ω and label $\lambda \in \mathcal{L}$, the function ϵ_λ^ω determines how the state evolves when the analysis traverses (in the backward direction) this edge. This definition depends on an extra operator $\mathcal{A}_{a,K}^\omega$

[7] The notation introduced here and below is not parameterized by P to keep it simple even if, strictly speaking, it should be.

$$\epsilon^{\omega}_{;}(\sigma) = \sigma$$

$$\epsilon^{\omega}_{l=e}(\sigma) = \begin{cases} \sigma \cup \mathcal{A}^{\omega}_{e,K} & \text{if } l \text{ is a pointer, where } K = \{k \mid (\mathrm{base}(l), k) \in \sigma\} \\ \sigma & \text{otherwise} \end{cases}$$

$$\epsilon^{\omega}_{l=\texttt{alloc}(\kappa,n)}(\sigma) = \sigma$$

$$\epsilon^{\omega}_{\texttt{free}(l)}(\sigma) = \sigma$$

$$\epsilon^{\omega}_{?e}(\sigma) = \sigma$$

$$\epsilon^{\omega}_{?\backslash\texttt{valid}(a)}(\sigma) = \begin{cases} \sigma \cup \mathcal{A}^{\omega}_{a,v} \cup \epsilon^{\omega}_{?\backslash\texttt{initialized}(\&l)}(\sigma) & \text{if } l = \mathrm{base}(a) \text{ is an lvalue} \\ \sigma \cup \mathcal{A}^{\omega}_{a,v} & \text{otherwise} \end{cases}$$

$$\epsilon^{\omega}_{?\backslash\texttt{valid\_read}(a)}(\sigma) = \begin{cases} \sigma \cup \mathcal{A}^{\omega}_{a,v} \cup \epsilon^{\omega}_{?\backslash\texttt{initialized}(\&l)}(\sigma) & \text{if } l = \mathrm{base}(a) \text{ is an lvalue} \\ \sigma \cup \mathcal{A}^{\omega}_{a,v} & \text{otherwise} \end{cases}$$

$$\epsilon^{\omega}_{?\backslash\texttt{initialized}(a)}(\sigma) = \sigma \cup \mathcal{A}^{\omega}_{a,i} \cup \epsilon^{\omega}_{?\backslash\texttt{valid}(a)}(\sigma)$$

$$\epsilon^{\omega}_{?(p_1 \wedge p_2)}(\sigma) = \epsilon^{\omega}_{?p_1}(\sigma) \cup \epsilon^{\omega}_{?p_2}(\sigma)$$

$$\epsilon^{\omega}_{?(p_1 \vee p_2)}(\sigma) = \epsilon^{\omega}_{?p_1}(\sigma) \cup \epsilon^{\omega}_{?p_2}(\sigma)$$

$$\epsilon^{\omega}_{?\neg p}(\sigma) = \epsilon^{\omega}_{?p}(\sigma)$$

$$\epsilon^{\omega}_{?(t_1 \odot t_2)}(\sigma) = \zeta^{\omega}_{t_1}(\sigma) \cup \zeta^{\omega}_{t_2}(\sigma) \qquad \text{where } \odot \in \{\equiv, \leq\}$$

$$\zeta^{\omega}_{e}(\sigma) = \sigma$$

$$\zeta^{\omega}_{\backslash\texttt{base\_addr(a)}}(\sigma) = \sigma \cup \mathcal{A}^{\omega}_{a,b} \cup \epsilon^{\omega}_{?\backslash\texttt{valid}(a)}(\sigma)$$

$$\zeta^{\omega}_{\backslash\texttt{block\_length(a)}}(\sigma) = \sigma \cup \mathcal{A}^{\omega}_{a,b} \cup \epsilon^{\omega}_{?\backslash\texttt{valid}(a)}(\sigma)$$

$$\zeta^{\omega}_{\backslash\texttt{offset(a)}}(\sigma) = \sigma \cup \mathcal{A}^{\omega}_{a,b} \cup \epsilon^{\omega}_{?\backslash\texttt{valid}(a)}(\sigma)$$

Figure 10: Over-approximating backward dataflow analysis.

defined for an address value a at the program point ω and $K \subseteq \{v, i, b\}$ by:

$$\mathcal{A}^{\omega}_{a,K} = \{(\mathrm{base}(x), k) \mid x \in \mathcal{A}^{\omega}(\mathrm{base}(a)), k \in K\}$$

In other words, $\mathcal{A}^{\omega}_{a,K}$ represents the set of all address values that may share the same base as a at program point ω, taken with some observation purpose of K. It is used to add new elements into an analysis state without forgetting possible aliases. For short, for any $k \in \{v, i, b\}$, $\mathcal{A}^{\omega}_{a,k}$ will denote $\mathcal{A}^{\omega}_{a,\{k\}}$. The definition of ϵ also uses a set of functions ζ similar to ϵ but labeled by a term instead of an edge label.

In most cases, the transition functions ϵ simply propagate the analysis state in a natural way (cf. Fig. 10). Let us consider in more detail the particular cases of assignments and memory-related constructs. For **`\valid`**`(a)` and **`\valid_read`**`(a)`, the purpose v is always added for a. Moreover, as we saw before (cf. lines 22–25 of Fig. 8),

checking the validity of a may require to check initialization of the corresponding address value base $\mathrm{base}(a)$. The monitoring of the initialization check is enabled by the corresponding recursive call. This is only necessary when $\mathrm{base}(a)$ is a left value (otherwise it is already the result of an address operator &, and reading this address cannot lead to an undefined behavior).

For edge label **\initialized**(a), the observation purpose i is added, and the same actions as for validity of a are performed by the corresponding recursive call. In particular, this call ensures that purpose v is added as well (to ensure that the block is recorded in the store). Similarly, for terms with memory-related functions that need block-level monitoring (cf. Fig. 2), the transition functions ζ add the observation purpose b and perform again the same actions as for validity of a by the corresponding recursive call. Notice that the recursive calls necessarily terminate for any address value a. Indeed, while the recursive call of $\epsilon^{\omega}_{?\backslash\mathtt{valid}(a)}$ can be invoked for the same address value a, the recursive call $\epsilon^{\omega}_{?\backslash\mathtt{initialized}(\&\mathrm{base}(a))}$ is invoked by $\epsilon^{\omega}_{?\backslash\mathtt{valid}(a)}$ only when $\mathrm{base}(a)$ is an lvalue, and in this case it adds an address operator and, therefore, cannot produce an lvalue infinitely many times.

Another specific case is an assignment $l = e$ of a pointer type. Indeed, a new alias can be created in this case, therefore all monitoring requirements for $\mathrm{base}(l)$ discovered in the following statements should be also applied to the aliased address e.

Soundness of the dataflow analysis. The presented pre-analysis provides (over-approximated) information indicating for each program memory block whether it is necessary to monitor it and in which way it should be done: just for validity (that leads to recording the corresponding memory block(s) in the store), or also for initialization (that requires to add initialization-related library primitives), and whether block-level monitoring is required (that determines if the block should be always recorded in the Patricia trie store).

The pre-analysis results are used by E-ACSL2C in the following way. Let s be a statement of P at program point $\omega \in \mathcal{V}$ and let σ be the resulting state computed by the pre-analysis for ω. If statement s can have an impact on validity (respectively, initialization) of a memory value a, E-ACSL2C should record this impact only if $(\mathrm{base}(a), v) \in \sigma$ (respectively, $(\mathrm{base}(a), i) \in \sigma$). Furthermore, E-ACSL2C should record this impact in the Patricia trie only if $(\mathrm{base}(a), b) \in \sigma$.

Soundness of the dataflow analysis can be stated in terms of paths in the control flow graph as follows.

Conjecture 1 (Soundness of the dataflow analysis) *Let $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{E}, \pi, \mathcal{I}, \mathcal{F})$ be the control flow graph of P, and let*

$$\omega_0 \xrightarrow{l_0} \omega_1 \xrightarrow{l_1} \ldots \omega_g \xrightarrow{l_g} \omega_{g+1} \xrightarrow{l_{g+1}} \ldots \omega_h \xrightarrow{l_h} \omega_{h+1} \xrightarrow{l_{h+1}} \ldots$$

be a path in $\mathcal{G}$ with $\omega_0 = \mathcal{I}$ and $0 \leq g < h$. Suppose that l_h is of the form $?p$, that predicate p includes memory-related constructs and that transition $\omega_g \xrightarrow{l_g} \omega_{g+1}$ has an impact on their evaluation at l_h (i.e. the effect of l_g on their evaluation at l_h is not completely overloaded by other l_k with $g < k < h$). Let σ be the resulting state computed by the pre-analysis for program point ω_g. Then

- *if p contains $\texttt{\valid}(a)$ or $\texttt{\valid\_read}(a)$, then* $(\mathrm{base}(a), v) \in \sigma$*;*
- *if p contains $\texttt{\initialized}(a)$, then* $(\mathrm{base}(a), v) \in \sigma$ *and* $(\mathrm{base}(a), i) \in \sigma$*;*
- *if p contains one of block-level memory-related constructs, then* $(\mathrm{base}(a), v) \in \sigma$ *and* $(\mathrm{base}(a), b) \in \sigma$.

In other words, the pre-analysis ensures that all necessary monitoring information is properly preserved whenever this information may have an impact on the evaluation of a memory-related construct along some execution path[8]. The notion of *having an impact on evaluation of a memory-related construct* is intuitively clear and is not formalized here. A complete formalization of E-ACSL2C optimized by the pre-analysis and the proof of its soundness are left as future work.

The presented analysis can be further improved in several ways. One improvement would be to stop monitoring the initialization of a single variable x before an assignment of x (since the variable is necessarily initialized by the assignment). It is also possible to improve Principle **P6** of Sec. 4: for predicates requiring byte-level monitoring, there is no need to check first the shadow memory store if the observation purposes of its argument include b.

6. Evaluation

Objectives. We have implemented the shadow-memory-based and the hybrid monitoring solution inside the memory monitoring library for E-ACSL2C. We have also implemented a pre-analysis similar to that presented in Sec. 5. We have evaluated the performances of the resulting instrumented code of the different versions on several benchmarks. Since we are not aware of any other tool supporting such a rich specification language for C code as E-ACSL, we compare E-ACSL2C with Valgrind tool [7], a well-known tool for memory safety (even if their objectives are quite different). Our aim is to study the following research questions:

RQ1 evaluate the hybrid memory store w.r.t. the earlier tree-based monitoring;

RQ2 evaluate the hybrid memory store w.r.t. the shadow-memory-based monitoring;

RQ3 evaluate the hybrid memory store w.r.t. Valgrind tool [7];

RQ4 evaluate the benefits of the pre-analysis.

[8]The considered paths are inspired by def-use paths, but remain different for several reasons: several assignments of separate cells of a C array a may simultaneously have an impact on the same $\texttt{\initialized}(a)$ predicate for the whole array a within the same path; validity of a pointer can be added and removed (e.g. through pointer (de-)allocation) several times along the same path; etc.

Experimental Protocol. The benchmarks are annotated C programs whose specifications contain byte-level memory-related predicates as well as other functional properties not related to memory. In order to be able to compare the combined solution with both the shadow memory store and the Patricia trie store used separately, the specification does not include block-level memory-related annotations since they are not supported by the shadow-memory-based store alone.

We perform in total several hundreds of experiments for more than 30 parameterized programs obtained from about 10 examples with different levels of specifications and different values of parameters. These initial experiments are conducted on small-size examples because they had to be manually specified in E-ACSL. To simulate longer execution, we choose higher values of parameters (such as matrix or array sizes) and execute programs several times for different input values. Execution time has been measured on an Intel Core i7-3520M 2.90GHz, 16GB of RAM.

These experiments have been conducted for the instrumented code without pre-analysis (where all memory locations are monitored) and with the reduced monitoring after the pre-analysis (described in Sec. 5). Fig. 11 and 13 present in detail some selected results. Fig. 11 shows execution time for the original non-instrumented code (column "Orig."), and for the instrumented program produced by E-ACSL2C without pre-analysis using the Patricia trie store alone ("PT"), the shadow memory store alone ("Sh"), and the hybrid solution ("H"). Similarly, Fig. 13 shows execution time for the instrumented program produced by E-ACSL2C with pre-analysis, as well as the time of analysis with Valgrind tool [7]. The columns "H vs PT" and "H vs Sh" indicate the speedup ($-N\%$) or slowdown ($+N\%$) recorded for the hybrid solution, respectively, w.r.t. the tree-based and the shadow-memory-based monitoring used separately. The speedups/slowdowns of Figures 11 and 13 are graphically represented in Figures 12 and 14 respectively, where the axes correspond to the columns "H vs PT" and "H vs Sh" and each point illustrates the speedups/slowdowns recorded for an example.

The peak residential set size (i.e. peak used memory size) for the same benchmarks is presented in Fig. 15 and Fig. 17. The shadow memory allocates a large set of virtual memory, of which only a small portion is actually used (proportional to the amount of memory used by the uninstrumented program). Similarly to execution time, the columns "H vs PT" and "H vs Sh" indicate reduction ($-N\%$) or increase ($+N\%$) in memory usage recorded for the hybrid solution w.r.t., respectively, the tree-based and shadow-memory-based monitoring used separately. Figures 16 and 18 graphically illustrate the "H vs PT" and "H vs Sh" columns (respectively, for Figures 15 and 17) by the coordinates of the points in the figures.

Experimental Results. First, the results show that the shadow-memory-based monitoring is indeed almost always faster than the Patricia trie, and can be dramatically faster (more than 90% speedup) on examples with frequent memory lookups and a big size of the Patricia trie. Interestingly, two exceptions are the `bubbleSort` example after the pre-analysis and the `binSearch` example, where the store contains very few memory locations, and most queries concern a very big array. In this case the lookup in a small tree becomes very fast, and even faster than accessing all the bytes in the shadow of a very big block. The decision to always store bigger blocks (longer than C bytes, cf. Principle P4) in the Patricia trie helps to preserve the better efficiency of the Patricia trie

Example	Orig.	E-ACSL2C without pre-analysis				
		PT	Sh	H	H vs PT	H vs Sh
bubbleSort	0.56	57.49	8.60	8.76	−84.76%	+1.86%
binSearch	0.00	2.91	6.74	2.87	−1.37%	−57.42%
mergeSort	0.04	106.94	0.47	0.45	−99.58%	−4.26%
quickSort	0.00	41.51	0.10	0.12	−99.71%	+20.00%
RedBlTree	0.03	0.73	0.20	0.29	−60.27%	+45.00%
merge	0.01	1.48	0.10	0.10	−93.24%	0.00%
matrixMult	0.13	4.32	1.23	1.22	−71.76%	−0.81%
matrixInv	0.01	4.29	1.77	1.79	−58.28%	+1.13%
insertSort	2.67	46.64	35.43	35.22	−24.49%	−0.59%

Figure 11: Execution time (in sec.) of the original program and the instrumented code after E-ACSL2C without pre-analysis: the hybrid memory monitoring (H) w.r.t. the Patricia trie store (PT) and the shadow memory store (Sh).

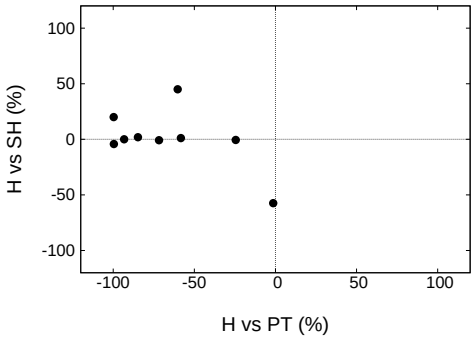


Figure 12: Execution time of the instrumented code after E-ACSL2C without pre-analysis: the hybrid memory monitoring (H) compared to the Patricia trie store (PT) and the shadow memory store (Sh). The coordinates of each point are the increase/decrease rates of the last two columns of Fig. 11 for an example.

in the hybrid store. The hybrid solution basically approaches the best of both separate kinds of monitoring, with a little additional cost (for initialization of a second store and determining which store must be used, cf. Sec. 4), that remains either below 2% or, on the fastest examples, below 0.1 sec. While the hybrid model does not always lead to a big speedup w.r.t. the shadow-memory-based store and can even become slightly slower, it significantly increases the expressiveness of the shadow-memory-based store by supporting block-level predicates and thus allows the user to mix byte- and block-level predicates in the same program.

In our examples, among several studied values, the values $C = 16$ or 32 seem to provide equally good results. They redirect to shadow memory most variables of primitive C types and leave in the Patricia trie bigger blocks (arrays, structures). However, we believe that the value of C can sometimes be even better adapted to the particular memory profile of the program under verification in order to make the hybrid store even more efficient. The study of optimal value for particular memory usage profiles is left as future work.

Regarding **RQ4**, the pre-analysis accelerates memory monitoring by removing ir-

Example	Val-grind	E-ACSL2C with pre-analysis PT	Sh	H	H vs PT	H vs Sh
bubbleSort	9.47	4.13	4.82	4.50	+8.96%	−6.64%
binSearch	0.48	2.85	6.68	2.90	+1.75%	−56.59%
mergeSort	2.68	86.28	0.43	0.42	−99.51%	−2.33%
quickSort	0.54	1.15	0.06	0.07	−93.91%	+16.67%
RedBlTree	2.29	0.59	0.19	0.28	−52.54%	+47.37%
merge	1.08	1.25	0.08	0.08	−93.60%	0.00%
matrixMult	1.85	3.46	0.67	0.68	−80.35%	+1.49%
matrixInv	0.70	3.45	1.59	1.66	−51.88%	+4.40%
insertSort	35.61	2.78	2.80	2.78	0.00%	−0.71%

Figure 13: Execution time (in sec.) of Valgrind [7] and the instrumented code after E-ACSL2C with pre-analysis: the hybrid memory monitoring (H) w.r.t. the Patricia trie store (PT) and the shadow memory store (Sh).

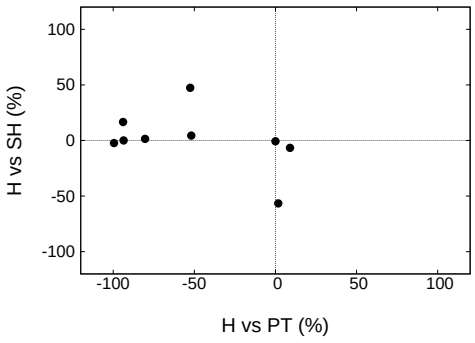


Figure 14: Execution time of the instrumented code after E-ACSL2C with pre-analysis: the hybrid memory monitoring (H) compared to the Patricia trie store (PT) and the shadow memory store (Sh). The coordinates of each point are the increase/decrease rates of the last two columns of Fig. 13 for an example.

relevant variables, and most often preserves the same ratios between the hybrid and separate monitoring, except the `bubbleSort` example where monitoring with the Patricia trie store becomes much faster after the pre-analysis. The global speedups provided by the pre-analysis are 60.22% for the Patricia trie store, 68.3% for the shadow-memory-based store and 73.69% for the hybrid store.

Regarding **RQ3**, we notice that Valgrind can be much faster (up to 6x) on some examples and much slower (up to 12.8x) on some others, that can be due to a different nature of properties evaluated by both tools (Valgrind looks for memory errors while E-ACSL2C checks specified annotations). For a more fair comparison with Valgrind where E-ACSL2C (with the hybrid store) and Valgrind address exactly the same properties, we have also performed another set of experiments on a different version of the same C benchmarks where the E-ACSL annotations specifically (and exclusively) focus on potential memory errors. The results of these experiments were similar: Valgrind can be much faster (up to 8x) on some examples and much slower (up to 25x) on some others, and lead to the same conclusion that execution time after E-ACSL2C is not comparable to Valgrind.

Example	Orig.	Memory usage of E-ACSL2C without pre-analysis				
		PT	Sh	H	H vs PT	H vs Sh
bubbleSort	1336 kB	1240 kB	7432 kB	9496 kB	+665.81%	+27.77%
binSearch	1308 kB	1308 kB	7732 kB	7708 kB	+489.30%	−0.31%
mergeSort	3712 kB	288756 kB	14008 kB	13988 kB	−95.16%	−0.14%
quickSort	1168 kB	140408 kB	7352 kB	7484 kB	−94.67%	+1.80%
RedBlTree	5656 kB	22276 kB	20084 kB	52904 kB	+137.49%	+163.41%
merge	16264 kB	204304 kB	38716 kB	38780 kB	−81.02%	+0.17%
matrixMult	2124 kB	2640 kB	12676 kB	11116 kB	+321.06%	−12.13%
matrixInv	2152 kB	2996 kB	12332 kB	8732 kB	+191.46%	−29.19%
insertionSort	1272 kB	1356 kB	7740 kB	7880 kB	+481.12%	+1.81%

Figure 15: Peak residential set size (in kilobytes) of the original program and the instrumented code after E-ACSL2C without pre-analysis: the hybrid memory monitoring (H) w.r.t. the Patricia trie store (PT) and the shadow memory store (Sh).

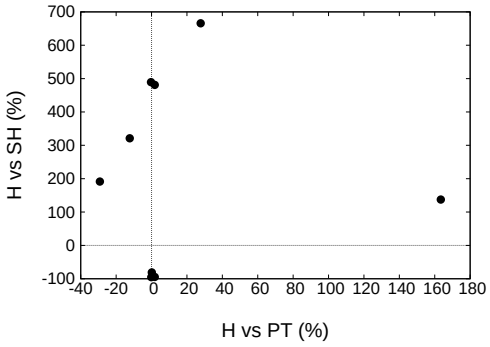


Figure 16: Peak residential set size of the instrumented code after E-ACSL2C without pre-analysis: the hybrid memory monitoring (H) compared to the Patricia trie store (PT) and the shadow memory store (Sh). The coordinates of each point are the increase/decrease rates of the last two columns of Fig. 15 for an example.

The slowdown of the code instrumented by E-ACSL2C w.r.t. the original code can be indeed higher than that of recent advanced tools that focus on memory safety errors. This is due to the larger scope of runtime assertion checking for E-ACSL: along with memory-related properties, it checks all other specified functional properties that lead to a lot of additional code inserted by the instrumentation to check the required properties (e.g. correct matrix multiplication or inversion, correct array sorting that globally preserves the same array elements, etc.). This additional code may include lots of loop iterations to verify universally quantified properties and often requires much more time than the original program code itself. This seems to be the price to pay for runtime assertion checking for an expressive specification language like E-ACSL.

Regarding memory usage (detailed in Fig. 15 and 17), despite a great theoretical amount of allocated virtual memory, the hybrid solution without pre-analysis shows on average only 3.24× increase of used memory peak (going in the worst-case up to 7.6581×, i.e. +665.81%) w.r.t. the Patricia trie, and on average only a 1.17× increase (going up to 2.63×) w.r.t. to the shadow-memory-based implementation. The hybrid solution with pre-analysis shows on average only 1.79× increase of used memory peak

Example	Val-grind	Memory usage of E-ACSL2C with pre-analysis				
		PT	Sh	H	H vs PT	H vs Sh
bubbleSort	46056 kB	1312 kB	5332 kB	5384 kB	+310.37%	+0.98%
binSearch	46420 kB	1372 kB	3280 kB	3344 kB	+143.73%	+1.95%
mergeSort	211128 kB	275028 kB	9812 kB	9836 kB	−96.42%	+0.24%
quickSort	46088 kB	91860 kB	5332 kB	5328 kB	−94.20%	−0.08%
RedBlTree	55916 kB	22248 kB	15916 kB	48816 kB	+119.42%	+206.71%
merge	175208 kB	204308 kB	36748 kB	36652 kB	−82.06%	−0.26%
matrixMult	46384 kB	2628 kB	6312 kB	6660 kB	+153.42%	+5.51%
matrixInv	46056 kB	3020 kB	8324 kB	10872 kB	+260.00%	+30.61%
insertionSort	46056 kB	1280 kB	1280 kB	1324 kB	+3.44%	+3.44%

Figure 17: Peak residential set size (in kilobytes) of Valgrind [7] and the instrumented code after E-ACSL2C with pre-analysis: the hybrid memory monitoring (H) w.r.t. the Patricia trie store (PT) and the shadow memory store (Sh).

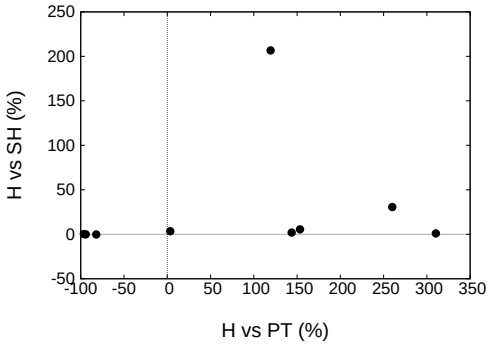


Figure 18: Peak residential set size of the instrumented code after E-ACSL2C with pre-analysis: the hybrid memory monitoring (H) compared to the Patricia trie store (PT) and the shadow memory store (Sh). The coordinates of each point are the increase/decrease rates of the last two columns of Fig. 17 for an example.

(going in the worst-case up to 4.1×) w.r.t. the Patricia trie, and on average only a 1.27× increase (going up to 3.06×) w.r.t. to the shadow-memory-based implementation. Interestingly, despite using two stores, the hybrid approach can sometimes even decrease the amount of used memory.

Valgrind uses on average 5.9× more memory than the hybrid approach without pre-analysis and 11.6× more memory than the hybrid approach with pre-analysis. Thus the hybrid solution turns out to provide better performance with a quite reasonable memory usage increase.

Summary of Experiments. The experiments confirm the expected benefits of the hybrid memory monitoring (cf. bounds (1) and (2) in Sec. 4), and of the pre-analysis. In particular:

RQ1 the hybrid memory monitoring is significantly faster than the tree-based memory monitoring, and does not imply any loss of expressiveness: it remains compatible with byte-level and block-level memory-related annotations;

RQ2 on byte-level memory annotations, the hybrid memory store remains comparable or slightly faster than the shadow-memory-based monitoring, and allows, in addition, the support of block-level E-ACSL predicates;

RQ3 the execution time of the code produced by E-ACSL2C is not comparable to Valgrind while the amount of used memory with Valgrind is considerably higher;

RQ4 the pre-analysis significantly improves the performance of the instrumented code.

Overall, our experiments suggest that the proposed hybrid solution reconciles the efficiency of the shadow memory with the expressiveness (and in some cases, a better efficiency) of a tree-based store, with an insignificant time overhead and an acceptable increase of memory usage. The proposed static analysis reduces irrelevant monitoring and further accelerates the execution of the instrumented code.

Competitions on Runtime Verification. An orthogonal evaluation of E-ACSL2C has been performed during international competitions on runtime verification organized annually since 2014 as part of the International Conference on Runtime Verification. E-ACSL2C participated in the Track on C Program Monitoring both at the 2014 and 2015 editions of the competition [24, 25]. In 2014, out of four tools initially registered for participation in the track and three tools having submitted their solutions for evaluation, E-ACSL2C arrived at the second place[9]. In 2015, out of six tools initially registered for the track and two tools having eventually submitted their solutions, E-ACSL2C arrived at the first place[10]. The evaluation criteria included soundness, memory consumption and runtime performance. Notably, the soundness scores of E-ACSL2C were particularly high: it was only 5% behind the winner in 2014, and more than twice higher than that of the second tool in 2015.

7. Related Work

The present work is part of an extension of FRAMA-C, an existing toolset for the 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 [26]. 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 [16], JML [13] for Java and Spark2014[11], a subset of Ada dedicated to formal testing and verification. Other examples of executable specification languages are SPEC# [27] and the closely related Code Contracts for .NET [28].

Since the main purpose of this paper is the support of memory-related E-ACSL annotations, this work is also related to previous efforts for ensuring memory safety of

[9] Final results at `http://rv2014.imag.fr/monitoring-competition/results.html`
[10] Final results at `https://www.cost-arvi.eu/?page_id=664`
[11] http://www.spark-2014.org

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 [29, 30, 31, 32, 33] and appeared well-adapted for most spatial errors (that is, accesses outside the bounds [34]). Advantages of these solutions 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 certain temporal errors (that is, accesses to an object that has been deallocated [34]). An alternative approach is based on pointer metadata stored inside multi-word *fat pointers* extending the pointer representation with bounds information [35, 36, 37]. While this approach to monitoring has the benefit of simplifying certain operations, for instance copying a pointer, it may complicate others such as pointer arithmetic. In addition, it can modify the memory layout of the program and complicate interfacing with external libraries. When an external function is called with a fat pointer as argument, it has to be converted to a regular "thin" pointer. For these reasons fat pointers were not deemed interesting for us. Techniques combining ideas of both approaches have been proposed as well [38, 34].

The technique of *shadow pages* [7, 8] 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. The usage of a Patricia trie for storing metadata in this context was proposed and evaluated in [5]. Despite several optimizations, this implementation still has a significant execution time overhead.

The present work continues the previous efforts and shows how the earlier Patricia trie store [5] and the efficient solutions based on shadow memory [7, 8] can be combined and adapted to our objective to support runtime assertion checking for a rich specification language as E-ACSL. We also give a rigorous presentation of a static analysis step in order to optimize the monitoring and determine the appropriate store to be used. To the best of our knowledge, such a combination of a Patricia trie with shadow memory for storing block metadata has never been studied. We show that this combination can significantly improve the performance of runtime assertion checking.

It should be noticed that the ambitious objective to perform runtime assertion checking for C code specified in E-ACSL and directly compatible with integrated FRAMA-C tools for proof of programs (as in [39]) can justify a higher overhead. Indeed, during deductive verification, manual analysis of proof failures without any automatic runtime checking could be even more costly.

8. Conclusion and Future Work

We have proposed an original hybrid memory monitoring solution that takes the best of two alternative monitoring techniques: a tree-based metadata storage in a Patri-

cia trie and an offset-based access to metadata in shadow memory. While a tree-based store is suitable to monitor both byte-level and block-level memory predicates, the shadow memory store is in general faster. Combining both solutions significantly improves the efficiency of the instrumented code, often providing a spectacular speedup w.r.t. the Patricia trie alone, and remains compatible with all memory-related E-ACSL predicates. A preliminary static analysis step allows to determine which memory locations should be monitored, and which store they should be registered.

Currently, the hybrid memory monitoring library is developed for 64-bit architectures only and is not yet fully integrated with a pre-analysis. The currently distributed version of pre-analysis already implements several representative features (including optimizations for validity and initialization monitoring, as well as an inter-procedural analysis not presented in this paper), while a better integration with the complete pre-analysis step is still under development. Another ongoing work is aimed at a better detection of some subtle temporal errors in E-ACSL, where we essentially use the possibility of shadow memory to use a scale and to store more than one byte of metadata for a byte of user memory.

Future work includes the extension of the proposed memory monitoring library to support a 32-bit architecture, a more efficient pre-analysis to identify memory locations for block-level and byte-level monitoring, proof of its soundness and further experiments to evaluate the solutions on bigger examples. Another future work direction is investigating beyond which size threshold C a big block should be redirected into the Patricia trie (cf. Principle P4). Finally, choosing the store to be used for a block according to the access frequency of the block metadata is another optimization heuristic to be studied.

Acknowledgments. This work has been partially funded by EU FP7 (project STANCE, grant 317753). The authors thank the FRAMA-C team members for support and useful discussions, and the anonymous referees for useful remarks and suggestions of improvement. Special thanks to Matthieu Lemerre for his advice on implementation of the shadow memory technique.

References

[1] M. Sullivan, R. Chillarege, Software defects and their impact on system availability: A study of field failures in operating systems, in: FTCS 1991, IEEE Computer Society, 1991, pp. 2–9.

[2] L. A. Clarke, D. S. Rosenblum, A historical perspective on runtime assertion checking in software development, ACM SIGSOFT Software Engineering Notes 31 (3) (2006) 25–37.

[3] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-C: A software analysis perspective, Formal Asp. Comput. 27 (3) (2015) 573–609.

[4] M. Delahaye, N. Kosmatov, J. Signoles, Common specification language for static and dynamic analysis of C programs, in: the 28th Annual ACM Symposium on Applied Computing (SAC 2013), ACM, 2013, pp. 1230–1235.

[5] N. Kosmatov, G. Petiot, J. Signoles, An optimized memory monitoring for runtime assertion checking of c programs, in: the 4th International Conference on Runtime Verification (RV 2013), Vol. 8174 of LNCS, Springer, 2013, pp. 167–182.

[6] W. Szpankowski, Patricia tries again revisited, J. ACM 37 (4) (1990) 691–711.

[7] N. Nethercote, J. Seward, How to shadow every byte of memory used by a program, in: the 3rd International Conference on Virtual Execution Environments (VEE 2007), ACM, 2007, pp. 65–74.

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

[9] A. Jakobsson, N. Kosmatov, J. Signoles, Fast as a shadow, expressive as a tree: hybrid memory monitoring for C, in: the 30th Annual ACM Symposium on Applied Computing (SAC 2015), ACM, 2015, pp. 1765–1772.

[10] J. Signoles, E-ACSL: Executable ANSI/ISO C Specification Language, http://frama-c.com/download/e-acsl/e-acsl.pdf (May 2015).

[11] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, V. Prevosto, ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html.

[12] Y. Cheon, A Runtime Assertion Checker for the Java Modeling Language, Iowa State Univ., 2003.

[13] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok, How the design of JML accomodates both runtime assertion checking and formal verification, in: FMCO 2002, Vol. 2852 of LNCS, Springer, 2002, pp. 262–284.

[14] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, N. Williams, CAVEAT: a tool for software validation, in: DSN 2002, IEEE Computer Society, 2002, p. 537.

[15] J.-C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification, in: CAV 2007, Vol. 4590 of LNCS, Springer, 2007, pp. 173–177.

[16] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, Inc., 1988.

[17] ISO/IEC 9899:1999, Programming languages – C.

[18] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, J. Signoles, Instrumentation of annotated C programs for test generation, in: Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on, IEEE, 2014, pp. 105–114.

[19] P. Herms, Certification of a tool chain for deductive program verification, Ph.D. thesis, University of Paris 11 (Dec. 2012).

[20] A. Giorgetti, J. Groslambert, J. Julliand, O. Kouchnarenko, Verification of class liveness properties with Java Modeling Language, IET Software 2 (6).

[21] L. Correnson, J. Signoles, Combining Analyses for C Program Verification, in: the 17th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012), Vol. 7437 of LNCS, Springer, 2012, pp. 108–130.

[22] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[23] B. Steensgaard, Points-to Analysis in Almost Linear Time, in: the 23rd Symposium on Principles of Programming Languages (POPL'96), ACM, New York, NY, USA, 1996, pp. 32–41.

[24] E. Bartocci, B. Bonakdarpour, Y. Falcone, First international competition on software for runtime verification, in: the 5th International Conference on Runtime Verification (RV 2014), Vol. 8734 of LNCS, Springer, 2014, pp. 1–9.

[25] Y. Falcone, D. Nickovic, G. Reger, D. Thoma, Second international competition on runtime verification CRV 2015, in: the 6th International Conference on Runtime Verification (RV 2015), Vol. 9333 of LNCS, Springer, 2015, pp. 405–422.

[26] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebr. Program. 78 (5) (2009) 293–303.

[27] M. Barnett, K. R. M. Leino, W. Schulte, The Spec# programming system: An overview, in: the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Int. Workshop (CASSIS 2004), Vol. 3362 of LNCS, Springer, 2004, pp. 49–69.

[28] M. Fähndrich, M. Barnett, F. Logozzo, Embedded contract languages, in: SAC 2010, ACM, 2010, pp. 2103–2110.

[29] R. W. M. Jones, P. H. J. Kelly, Backwards-compatible bounds checking for arrays and pointers in c programs, in: the Third International Workshop on Automatic Debugging (AADEBUG 1997), 1997, pp. 13–26.

[30] O. Ruwase, M. S. Lam, A practical dynamic buffer overflow detector, in: NDSS 2004, 2004, pp. 159–169.

[31] W. Xu, D. C. DuVarney, R. Sekar, An efficient and backwards-compatible transformation to ensure memory safety of C programs, in: FSE 2004, ACM, 2004, pp. 117–126.

[32] D. Dhurjati, V. S. Adve, Backwards-compatible array bounds checking for C with very low overhead, in: ICSE 2006, 2006, pp. 162–171.

[33] P. Akritidis, M. Costa, M. Castro, S. Hand, Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors, in: USENIX 2009, USENIX Association, 2009, pp. 51–66.

[34] M. S. Simpson, R. Barua, MemSafe: ensuring the spatial and temporal memory safety of C at runtime, Softw., Pract. Exper. 43 (1) (2013) 93–128.

[35] T. M. Austin, S. E. Breach, G. S. Sohi, Efficient detection of all pointer and array access errors, in: PLDI 1994, ACM, 1994, pp. 290–301.

[36] G. C. Necula, J. Condit, M. Harren, S. McPeak, W. Weimer, CCured: type-safe retrofitting of legacy software, ACM Trans. Program. Lang. Syst. 27 (3) (2005) 477–526.

[37] Y. Oiwa, Implementation of the memory-safe full ANSI-C compiler, in: PLDI 2009, ACM, 2009, pp. 259–269.

[38] J. Yuan, R. Johnson, CAWDOR: compiler assisted worm defense, in: SCAM 2012, IEEE Computer Society, 2012, pp. 54–63.

[39] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, J. Julliand, Your proof fails? testing helps to find the reason, in: the 10th International Conference on Tests and Proofs (TAP 2016), Vol. 9762 of LNCS, Springer, 2016, pp. 130–150.