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

Arvid Jakobsson[1] Nikolai Kosmatov[2] Julien Signoles[2]
[1,2]CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
[1]firstname.lastname@gmail.com, [2]firstname.lastname@cea.fr

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 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 analysis of C code. We present an original combination of a tree-based and a shadow-memory-based techniques that reconciles both the efficiency of shadow memory with the higher expressiveness of annotations whose runtime evaluation can be ensured by 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. This combined monitoring technique has 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.

1. INTRODUCTION

Over the past few decades, memory safety of C programs has been addressed in numerous research works 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. They 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 [25] 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 [7]. Runtime checking of memory-related properties can be realized using systematic monitoring of memory operations. However, to do so efficiently is difficult, due to 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 [8], a platform for 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 [9]. 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 [14] records memory block metadata in a compact prefix tree (Patricia trie) [26], 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 [14], 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

*This work has been partially funded by EU FP7 (project STANCE, grant 317753).

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2695664.2695815

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

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

metadata needed, and thus several memory accesses.

Recent advanced tools for 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 [19, 22]. Such an array is called a *shadow memory*, since each address of the user-memory is shadowed by an element of this array. 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.

The objective of this paper is to study how the existing 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 both the efficiency of shadow memory with the higher expressiveness of annotations whose runtime evaluation can be ensured by a tree of metadata. Rather than providing detailed (but more difficult to follow) algorithms, we give comprehensive design principles of the combined technique. We implement these techniques in the memory monitoring library for runtime assertion checking in FRAMA-C [8] and evaluate them on several experiments.

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,
- rough complexity evaluation for the Patricia trie store and the shadow memory store,
- an original hybrid memory monitoring solution combining two kinds of metadata storage,
- implementation of the combined monitoring for E-ACSL2C,
- evaluation of the hybrid solution compared to separate monitoring using a Patricia trie and shadow memory.

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

2. RUNTIME ASSERTION CHECKING OF E-ACSL ANNOTATIONS

Overview of E-ACSL.

This section presents E-ACSL [9, 23], an executable specification language designed to support runtime assertion checking[1] in FRAMA-C.

FRAMA-C [8] is a framework dedicated to 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 [4] is a behavioral specification language shared by different FRAMA-C analyzers that takes the best of the specification languages of earlier tools CAVEAT [5] and CADUCEUS [12], inspired by JML [6, 15].

ACSL is sufficiently expressive to express most functional properties of C programs. It has already been used in many projects, including large-scale industrial ones [8]. 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 [17] may be associated to each function in order to specify its pre- and postconditions. The contract can be split into several named guarded behaviors. Contracts may also be associated to statements, as well as assertions, loop invariants and loop variants. ACSL annotations also include definitions of (inductive) predicates, axiomatics, lemmas, logic functions, data invariants and ghost code.

Designed as a large subset of ACSL, E-ACSL preserves ACSL semantics. Moreover, the E-ACSL language is *executable*: its annotations can be translated into C monitors 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). There is also no way to express termination properties of a loop or a recursive function, since 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.

The first two columns of Fig. 1 present some memory-related annotations supported by E-ACSL. 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.

Running Example.

Fig. 2 contains a toy example illustrating memory-related predicates of E-ACSL. The code at lines 6–19 checks if the array of in-

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

```
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   else
19     result = (void*)0;    // not found, NULL
20   // check the result
21   //@ assert \base_addr(arr) == \base_addr(result);
22   //@ assert \offset(arr) <= \offset(result) <= \offset(arr)+(len-sublen)*sizeof(int);
23   //@ assert \forall int j; 0 <= j < sublen ==> result[j] == subarr[j];
24 }
```

Figure 2: 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

tegers `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. 2 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 21–23 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 21), that `result` points to an element inside the appropriate part of `arr` (line 22) and that this element starts indeed an occurrence of the subarray `subarr` (line 23).

Runtime Assertion Checking for E-ACSL.

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 of C code by E-ACSL2C as described in [9]. However, runtime assertion checking of E-ACSL specifications involving memory-related constructs of Fig. 1 requires particular care.

In order to evaluate memory-related E-ACSL annotations (cf. Fig. 1), we record metadata on validity, initialization, size, etc. of memory locations during program execution in a dedicated data store, that we call below *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 equal to 8, the predicate `\valid(arr+i)` at line 4 of Fig. 2 is translated by E-ACSL2C 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 [14].

Pre-Analysis for Limited Monitoring.

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 comes with 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. This analysis has been presented in [9]. For example, to evaluate the annotations of the program in Fig. 2 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 23). 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.

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.

3. HYBRID MEMORY MONITORING

Byte-Level and Block-Level Monitoring.

Memory-related annotations of E-ACSL presented in Fig. 1 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. 1). 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. 2 include only byte-level predicates, while the assertions at lines 21–22 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 `\valid(arr)` requires block-level monitoring because of the assertions at lines 21–22.

Let us now present a tree-based and a shadow-memory-based storage of block 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 [14] proposed a memory monitoring solution for E-ACSL based on a compact prefix tree (Patricia trie) [26] 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. 3a 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. 3b 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. 3b would give that of Fig. 3a.

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 [14], 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, initialization, 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* [19, 22], 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 of a user memory address p to the shadow memory is basically a linear mapping:

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

obtained by a simple offset with a scale (if more, or less, than one byte of shadow memory should be associated to each byte of the user memory).

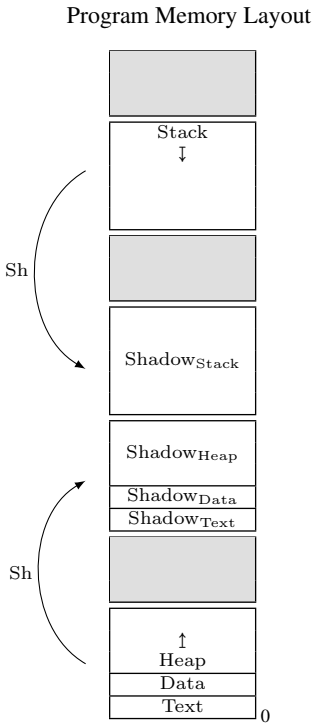

Figure 4: Metadata storage based on shadow memory

The design of the new shadow-memory-based store for memory monitoring with E-ACSL is illustrated in Fig. 4. 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. 4 show memory zones that are unaddressable and/or not represented by the shadow memory store. An access to the metadata for an ad-

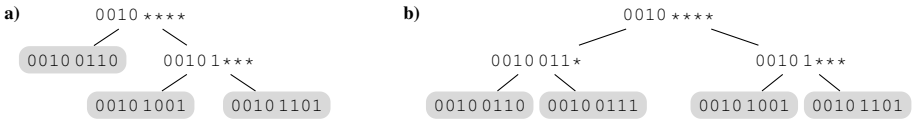

Figure 3: Example of a compact prefix tree (Patricia trie) a) before, and b) after inserting `0010 0111`

dress 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 strong and 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

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

more 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, unless 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.

In addition, there are some specific cases and optimizations.

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. 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.

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.

4. EVALUATION

Objectives.

We have implemented the shadow-memory-based and the hybrid monitoring solution inside the memory monitoring library for E-ACSL2C, and evaluated them on several benchmarks. Our aim is to study the following research questions:

RQ1 evaluate the hybrid memory store w.r.t. the earlier tree-based memory 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 [19].

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. Experiments are performed on an Intel Core i7-3520M 2.90GHz, 16GB of RAM.

Fig. 5 presents in detail some selected results. Its columns show the execution time for the original non-instrumented code (column "Orig."), for the instrumented program produced by E-ACSL2C using the Patricia trie store alone ("PT"), the shadow memory store alone ("Sh"), and the hybrid solution ("H"). These experiments have been conducted for instrumented code without pre-analysis (where all program memory locations are monitored) and with the reduced monitoring after the pre-analysis (described in Sec. 2). The columns "H vs PT" and "H vs Sh" indicate the speedup ($-N\%$) or slowdown ($+N\%$) recorded for the hybrid solution w.r.t., respectively, the tree-based and shadow-memory-based monitoring used separately. Time of analysis with Valgrind tool [19] is indicated in the last column.

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 and the `binSearch` example after the pre-analysis, 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 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. 3), that remains either below 2% or, on the fastest examples, below 0.1 sec.

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 to make the hybrid store even more efficient. The study of optimal value for particular memory usage profiles is left as future work.

The pre-analysis accelerates memory monitoring by removing irrelevant 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.

Finally, notice that execution time after E-ACSL2C is not comparable to Valgrind. Valgrind can be much faster (up to 6x) on some examples and much slower (up to 12.8x) on some others, that can be partially due to the pre-analysis that strongly reduces monitored variables (e.g. example `insertSort`).

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 (not detailed in Fig. 5), despite a great theoretical amount of allocated virtual memory, the hybrid solution shows in average only 3.5× increase of *used memory peak* (with the worst-case increase up to 8.8×) w.r.t. the Patricia trie, and in average only a 1.3× increase (going up to 2.8× in the worst case) w.r.t. to the shadow-memory-based implementation. Valgrind uses in average 22.9× more memory than the hybrid approach. Thus a combination with shadow memory 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. 3). In particular:

RQ1 the hybrid memory monitoring is significantly faster than the

Example	Orig.	Without pre-analysis					With pre-analysis					Val-grind
		PT	Sh	H	H vs PT	H vs Sh	PT	Sh	H	H vs PT	H vs Sh	
bubbleSort	0.56	57.49	8.60	8.76	−84.76%	+1.86%	4.13	4.82	4.50	+8.96%	−6.64%	9.47
binSearch	0.00	2.91	6.74	2.87	−1.37%	−57.42%	2.85	6.68	2.90	+1.75%	−56.59%	0.48
mergeSort	0.04	106.94	0.47	0.45	−99.58%	−4.26%	86.28	0.43	0.42	−99.51%	−2.33%	2.68
quickSort	0.00	41.51	0.10	0.12	−99.71%	+20.00%	1.15	0.06	0.07	−93.91%	+16.67%	0.54
RedBlTree	0.03	0.73	0.20	0.29	−60.27%	+45.00%	0.59	0.19	0.28	−52.54%	+47.37%	2.29
merge	0.01	1.48	0.10	0.10	−93.24%	0.00%	1.25	0.08	0.08	−93.60%	0.00%	1.08
matrixMult	0.13	4.32	1.23	1.22	−71.76%	−0.81%	3.46	0.67	0.68	−80.35%	+1.49%	1.85
matrixInv	0.01	4.29	1.77	1.79	−58.28%	+1.13%	3.45	1.59	1.66	−51.88%	+4.40%	0.70
insertSort	2.67	46.64	35.43	35.22	−24.49%	−0.59%	2.78	2.80	2.78	0.00%	−0.71%	35.61
Total	3.47	266.31	54.64	50.82	−80.92%	−6.99%	105.94	17.32	13.37	−87.38%	−22.81%	54.7

Figure 5: The hybrid memory monitoring (H) w.r.t. the Patricia trie store (PT), the shadow memory store (Sh) and Valgrind

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 and depends on the nature and amount of evaluated E-ACSL annotations, while the amount of used memory with Valgrind is considerably higher.

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.

5. RELATED WORK

The present work is part of an extension of FRAMA-C, an existing toolset for analysis of C code, for supporting runtime assertion checking. It is therefore related to a lot of works on runtime assertion checking [7] and, more generally, runtime verification [16]. 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 [17], JML [15] for Java and Spark2014[2], a subset of Ada dedicated to formal testing and verification. Other examples of executable specification languages are SPEC# [3] and the closely related Code Contracts for .NET [11].

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 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 [13, 21, 27, 10, 1] and appeared well-adapted for most spatial errors (that is, accesses outside the bounds [24]). 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 [24]). An alternative approach is based on pointer metadata stored inside multi-word *fat pointers* extending the pointer representation with bounds information [2, 18, 20]. 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 [28, 24].

The technique of *shadow pages* [19, 22] 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 [14]. 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 [14] and the efficient solutions based on shadow memory [19, 22] can be combined and adapted to our objective to support runtime assertion checking for a rich specification language as E-ACSL. 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 can justify a higher overhead. Indeed, during deductive verification, manual analysis of proof failures without any automatic runtime checking could be even more costly.

6. 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 Patricia 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 pred-

[2]http://www.spark-2014.org

icates, 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.

Currently, the hybrid memory monitoring library is developed for 64-bit architectures only and is not yet fully integrated with a pre-analysis to determine memory locations for block-level monitoring. A simple pre-analysis for block-level monitoring can basically be realized as an extra analysis phase similar to that explained in [9], while a more efficient pre-analysis is 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, finalizing a more efficient pre-analysis to identify memory locations for block-level and byte-level monitoring, 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. The authors thank the FRAMA-C team members for support and useful discussions. Special thanks to Matthieu Lemerre for his advice on implementation of the shadow memory technique. The authors also thank the anonymous reviewers for their suggestions of further improvements and future work.

7. REFERENCES

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

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI 1994*, pages 290–301. ACM, 1994.

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

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

[5] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. CAVEAT: a tool for software validation. In *DSN 2002*, page 537. IEEE Computer Society, 2002.

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

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

[8] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C, a program analysis perspective. In *SEFM 2012*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.

[9] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *SAC 2013*, pages 1230–1235. ACM, 2013.

[10] D. Dhurjati and V. S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE 2006*, pages 162–171, 2006.

[11] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC 2010*, pages 2103–2110. ACM, 2010.

[12] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV 2007*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[13] R. W. M. Jones and 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)*, pages 13–26, 1997.

[14] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of c programs. In *RV 2013*, volume 8174 of *LNCS*, pages 167–182. Springer, 2013.

[15] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. In *FMCO 2002*, volume 2852 of *LNCS*, pages 262–284. Springer, 2002.

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

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

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

[19] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE 2007*, pages 65–74. ACM, 2007.

[20] Y. Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *PLDI 2009*, pages 259–269. ACM, 2009.

[21] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS 2004*, pages 159–169, 2004.

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

[23] J. Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language*. URL: http://frama-c.com/download/e-acsl/e-acsl.pdf.

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

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

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

[27] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE 2004*, pages 117–126. ACM, 2004.

[28] J. Yuan and R. Johnson. CAWDOR: compiler assisted worm defense. In *SCAM 2012*, pages 54–63. IEEE Computer Society, 2012.