Automated Memoization for Parameter Studies Implemented in Impure Languages

Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University
{stoffers; schemmel; soriadustmann; wehrle}@comsys.rwth-aachen.de

ABSTRACT

In computer simulations many processes are highly repetitive. These repetitions are amplified further when a parameter study is conducted where the same model is repeatedly executed with varying parameters, especially when performing multiple runs to increase statistical confidence. Inevitably, such repetitions result in the execution of identical computations, with identical code, identical input, and hence identical output. Performing computations redundantly wastes resources and the execution time of a parameter study could be reduced if the redundancies were avoided.

To this end, the idea of memoization was proposed decades ago. However, until today memoization is either performed manually or automated memoization approaches are used that can only handle pure functions. This means that only the function parameters and the return value may be input and output of the function whereas side effects are not allowed. In order to expand the scope of automated memoization to a larger class of programs, we propose an approach able to reliably detect the full input and output of a function, including reading and writing objects through arbitrarily indirect pointers with some preconditions. We show the feasibility of our approach and derive simple performance approximations enabling rough predictions of the expected benefit. By means of a simple case study performing an OFDM network simulation, we demonstrate the practical suitability of our approach, speeding up the execution of the whole parameter study by a factor of 75, while only doubling memory consumption.

CCS Concepts

•**Computing methodologies → Massively parallel and high-performance simulations;** •**Software and its engineering** → *Preprocessors;*

Keywords

Automatic Memoization; Accelerating Parameter Studies; Impure Languages

SIGSIM-PADS '16, May 15 - 18, 2016, Banff, AB, Canada
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3742-7/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901378.2901386

1. INTRODUCTION

Computer simulations are programs with highly repetitive computations. While a simulation is running, the same event handlers are executed repeatedly, often on the same input as before. In order to achieve sufficient statistical confidence, simulation experiments are repeated with the same parameters, but different random number streams. This inevitably results in performing a certain subset of computations again. Finally, to compare different configurations, certain parameters are varied while others are kept constant. This again causes a large set of computations to be repeatedly performed on the same input, necessarily resulting in the same output. From this observation we conclude that in a simulation parameter study a large fraction of the computations performed are in fact redundant. The opportunity to speed up the execution of computer software in general by avoiding such redundant computations has already been described by Michie in 1968 [16]. Michie developed the idea of so called memo functions (now known as *memoized functions*), which allow re-using previously computed results.

In order to apply this technique, two steps have to be performed: 1. The code blocks have to be identified, which are executed redundantly and comprise computations with a complexity greater than the memoization overhead. 2. Those code blocks have to be transformed into a variant that stores input and output of a computation and is able to directly reproduce the correct output if the same input re-occurs. In current practice, both steps are often applied manually.

While it would naturally be desirable to automate both steps, in this paper we focus on the second step. This is especially motivated by the observation that model developers are often domain experts in the simulated field, but not necessarily programming experts as well. We argue that the knowledge of the developed simulation model allows them to identify redundant computations. However, applying memoization manually requires to fully understand the underlying concepts. The developer has to carefully identify all side effects of a function and then create the code that detects whether the input has been seen before, applies the results, or computes and stores them. Once realized for a specific problem, this code can, however, not be reused for a different problem, requiring a repetition of much of this labor-intense and error-prone effort. Hence, we target an environment where the developer only needs to perform the first step and annotate the promising code blocks, saving the effort of manually applying the memoization.

We discuss previously developed techniques to tackle this second step (called *automated memoization*) in more detail in Sect. 5. We conclude that, unfortunately, all of them are designed for pure functions[1], whose availability is often supported by properties of the target programming language, especially prevalent amongst functional programming languages. However, most simulation models are not written in functional programming languages and computations heavily rely on the features that allow writing impure code. Hence, techniques restricted to pure functions can not be applied to a large set of simulation models.

We conclude that till today there is no approach to *automated memoization for impure functions*. We argue that such a technique could significantly improve performance of complex software with redundant operations, as common in simulation parameter studies, without requiring much effort or advanced programming skills from the developer. We propose an approach to automated memoization, which does not rely on the purity of the code to memoize. Our implementation operates on C++ [9], which is used to implement models for the popular open source simulation frameworks ns-3 [5] and OMNeT++ [22], and provides many features that pose challenges for memoization (such as pointers), hence we expect that our approach can be easily adapted to other languages with a less challenging feature set. We do not require that the computation to be memoized is pure, and in fact we allow using arbitrarily indirect pointers to read objects or cause side effects. However, it is another result of this work that it is necessary to pose certain restrictions on pointer usage, for example, to avoid undecidable problems like static aliasing analysis [12] (see Sect. 3.6).

To apply the automated memoization, we parse the provided C++ code using Clang[2], identify all input and output, and finally generate new C++ code with a memoized version of the original code. This can then be compiled by any C++14 compiler. On execution of our memoized code a lookup in a dictionary (the *Memoization Cache, MC*) is performed. On success the result is applied to the actual output, otherwise the result is computed as in the original code, intercepted, and stored in the MC. Hence, our approach works on pure *and* impure computations.

The remainder of this paper is structured as follows. We first analyze the problem and the resulting challenges more thoroughly (Sect. 2). After that, we introduce the design of our approach in detail and discuss its capabilities and limitations (Sect. 3). We demonstrate the practical feasibility by presenting evaluation results (Sect. 4). We then discuss related work (Sect. 5) before we conclude the paper (Sect. 6).

2. PROBLEM ANALYSIS

The major challenge of memoization is the correct identification of input and output. While this is an easy task for pure functions, impure functions can traverse the object graph arbitrarily and access any element. We need to determine which values are actually input or output of the computation and which are just used to find the finally relevant object in memory, but whose own value has no semantical meaning to the computation (e. g., a pointer that is being dereferenced to access another object).

[1]A *pure* function accesses no objects except compile-time constants, its parameters, and its local variables with automatic storage duration. Its parameters and return type are of value type and it never throws exceptions. It inspects no pointers and calls only pure functions.

[2]http://clang.llvm.org/

In this section, we first investigate adequate levels of granularity for automated memoization. We then discuss the language constructs which can conceivably be memoized, analyze the features of C++, and derive the implications on automated memoization for a representative subset.

2.1 Memoization Granularity

All existing approaches to automated memoization apply the optimization on a function level, i. e., a function is either memoized as a whole, or not at all. As these approaches rely on the input being solely the function parameters and the output being exactly the return value, a function level granularity is the only viable approach. However, as we allow side effects, we need to analyze the actual implementation anyway and the restriction to functions is no longer useful.

For this reason, we allow the memoization of (almost) arbitrary C++ *compound-statements*, better known as *blocks*. Blocks are the most general construct that has a concept of local variables with automatic storage duration and encapsulates them from the outer environment. Any statement that can be wrapped in curly braces to form a block without changing the semantics of the program can also be a memoization target. To this end, memoizing a function is performed by memoizing the block that constitutes its body. Similarly, a complete event handler could be memoized since an event handler is typically a function as well.

In general, the unit to be memoized should be a logical unit of the program's functionality. If two computations were memoized as a single unit, it is highly unlikely that the results can be reused as the input of both computations must have occurred before in that combination. On the other hand, if the memoization unit ends before a computation is complete, several intermediate results have to be retrieved rather than the final result. In simulations an event handler can be a good memoization target if it performs a single computation. However, a specific computation performed inside the event handler can as well be a more promising memoization target. The remainder of this analysis assumes that a C++ compound statement (block) shall be memoized, referred to as the *Memoization Unit (MU)*.

2.2 Variable Scope

Our first observation is that any object that is both created and destroyed inside the MU can obviously be neither input nor output. From this observation we deduce that it is helpful to distinguish variables by their scope and lifetime.

C++ allows a multitude of different scopes, from file-level variables, over class members to variables with block scope. However, for a variable which is not local to the MU or has a storage duration other than automatic (e. g., static) we observe: If it is read inside the MU without a prior write operation, its value is input of the computation. If it is written inside the MU, the effects are visible after completion of the MU, hence the variable must be considered output. Only if a variable v that is accessed inside the MU is also defined inside with automatic storage duration, the computation can neither depend on v nor can a result be stored in v.

We conclude: If a variable is local to the MU and has automatic storage duration, it is neither input nor output. In all other cases, read and write operations to that variable have to be analyzed and the variable has to be considered potential input and/or output. In the following, we use the terms *interior* for the first and *exterior* for the second case.

2.3 Pointers

For detecting input and output of a function, the most challenging feature of C++ concerns pointers as it is oftentimes impossible to statically predict which object a given pointer will point to at runtime. We identify the following important operations that can be performed on pointers:

- Pointers can be copied (e. g., `p=...` or `...=p`).
- A pointer can be dereferenced to alter the object it points to (e. g., `*p=...`).
- A pointer can be dereferenced to read the object it points to (e. g., `...=*p`).
- Instead of accessing the object the pointer points to, the address can also be used to access neighboring objects (e. g., `*(p-4)` or `p[3]`).
- Any object, including those of pointer type, can have its address taken (e. g., `&p`).
- Other notable operations on pointers are comparisons and conversions (e. g., `!p` or `p!=nullptr`).
- If the object a pointer points to is a pointer itself, the definition is recursive (e. g., `*p[0]=****(***q+4)`).

To evaluate a computation's actual input and output, we need to investigate the semantics of these operations. The most useful operations on pointers are those ending up dereferencing them to access another object. Consider `*p=42`. A simplistic approach would identify two memory accesses: First, the pointer object is read. Then `42` is stored to the target object. Hence, the pointer's value would be identified as input, inhibiting memoization in many common cases.

We can circumvent this problem by respecting the semantics of this operation, which is to modify the object located at the address `p` points to. Hence, the actual *value* of `p` (i. e., the address) is irrelevant. A smart approach to automated memoization has to store the *path* an object is accessed through and only the object itself should be considered input or output. For `*p=42` it is hence necessary to store that the object pointed to by `p` is assigned 42.

We define such a path as an $(n+1)$-tuple where n is the number of dereferenced pointers along that path. The first component of this tuple is the pointer name, followed by the difference between the current pointer value of the path component and the address of the object that is actually accessed. Hence, `*p` is simply represented by the tuple ("p",0). We represent expressions like `a.b` as ("a","b") which allows us to follow the C++ standard in representing `a->b` and `(*a).b` as the path ("a",0,"b") while easily retaining type information even in the case of `unions`.

This also allows representing arbitrarily complex pointer expressions like `(p+4)[3]=q[8][-5][3]`: The object at ("q",8,-5,3) is read and the object at ("p",4,3) is altered.

Similarly, we represent taking of addresses by adding the special value `&` to this notation to indicate the addressof operation. Hence, e. g., `(&a)[3]` is represented by ("a",&,3).

As these semantics imply that the actual value of a pointer is unimportant, we only allow conversions to `bool` (implicit, explicit, or by comparison to null pointer constant), which we represent by adding the special value `bool` to the path. For example, the expression `p ? *p : 0` performs a read of ("p",`bool`) potentially followed by a read of ("p",0).

With respect to pointer comparisons two major cases have to be distinguished. Equality comparisons just test if the pointers point to the same object irrespective of the actual address. Ordered comparisons on the other hand are only specified in C++ if both pointers point to subobjects within the same superobject, e. g., elements of the same array. This means that, again, the actual pointer value is unimportant, as in fact the offsets within the superobject are being compared. All comparisons are encoded in our path notation as the components of the path to the left hand pointer, followed by a special symbol representing the comparison type, and finally the components of the path to the right hand pointer, e. g., `p<q` would be encoded as ("p",<,"q").

Remark: We use this path notation not only as a theoretical concept but as well in the implementation of our approach. Hence, we need a value for the name of an object. For convenience, we simply use a string representing the name of the object throughout this paper. However, as 1. even when using fully qualified names this does not allow variables in sibling scopes or anonymous namespaces, and 2. string processing is inefficient, our transformation deterministically assigns each variable it encounters a unique, numerical ID. If, e. g., the symbol `p` is assigned the ID 42, the object `p[3]` is described as (42,3). We would like to stress the importance that this assignment be deterministic and independent of the surrounding context, to ensure that the One Definition Rule is not accidentally violated.

2.4 References, Arrays, and Containers

Similarly to pointers, reference types can be used in C++ to alias objects. However, references are not more expressive than pointers, i. e., everything that can be expressed by a reference could be expressed by a pointer as well. Hence, we can treat references as pointers with a different syntax.

Indexing C-style arrays needs no special handling, as `a[1]` is equivalent by definition to `*(a+1)`. Hence, array-to-pointer conversion actually applies the indexing to a pointer.

Dealing with containers from the C++ standard library requires additional consideration. As the definition of a template such as `std::array` has to be available at compile time, it seems, at first glance, easy to consider class templates to be equivalent to user code. However, the C++ standard allows implementations significant leeway with respect to how these containers are actually implemented. One example of non-obvious optimizations is the use of SCARY iterators [20] to reduce the generated code size. Similarly, `std::vector` could conceivably be specialized for pointer-to-object types to always use the same non-generic implementation that is only available as a pre-compiled library with a C interface. We believe that in future efforts many of those operations can be serialized by making their semantics known during the memoization procedure. However, this is not important to demonstrate the general feasibility of automated memoization for impure functions.

2.5 Function Calls

If a function is called inside the MU, we need to differentiate two cases: If the function implementation is known to the compiler, it can be included into the analysis. However, this is not provided in general, e. g., for pre-compiled libraries. Such functions might or might not have side effects and might or might not depend on additional input. As this cannot be determined for functions whose implementation is unknown, we enable the user to annotate function signatures or calls as *transparent*. To this end, we define a transparent function as a function whose effects depend only on the parameters and are only of the following kinds: If a parameter of pointer or reference to a non-const object type is provided (e. g., ~~const~~ `int *p`), the value of the object may be changed during execution of the function (e. g., `*p=42`). The function may return an object or throw an exception. Other effects may occur if the user declares them negligible, e. g., logging or allocating temporary dynamic storage. As input we treat the parameter's values, or, for arguments of pointer or reference type, the object they point to.

For non-transparent functions whose implementation is known we need to differentiate whether the call is recursive. Non-recursive calls can just be handled like inline code. For recursive calls a fixpoint iteration is necessary to determine the full input and output. Since iterative programming is by far more commonly used in C++, we focus however on iterative functions and leave the fixpoint analysis of non-transparent, recursive function calls for future work.

Investigating the different methods to define and call functions in C++, we observe that all of them boil down to the simple base case. Member functions are just functions with an additional argument. For function pointers and virtual functions, either the pointer / call has to be annotated or the compiler needs to be able to statically determine which function is called and which code is executed. Lambda functions and function templates are just different ways to create functions. For implicit function calls (constructors, destructors, user defined conversions, and operators) the compiler can determine which function is executed and handle the implicit call just like an explicit call to that function. Hence, all kinds of functions can be treated as described above.

2.6 Unstructured Control Flow

C++ provides keywords that allow entering and leaving the MU not only at the beginning or end. Early exits (`return`, `break`, `continue`, `throw`, `goto`) have to be considered by the automated memoization as follows: Reads after the exit must not be considered input. While computing the results it has to be ensured that the output is stored in the MC prior to the early exit. When a result is retrieved from the MC, we need to ensure that the MU is left exactly the same way the original execution would leave the MU, including the correct argument to `return` or `throw`. This can be achieved by storing the kind of exit and potentially its argument as part of the output of the function.

Entering the MU at a point other than the beginning can only be achieved by jumping to a label in the MU. Though this can be allowed by extra care, we do not consider it in this paper as `goto` is not recommended anyway [1, 11] and memoizing a block that contains a label from a `switch` statement outside the MU is similarly discouraged.

As early *termination* (which, as opposed to early *exits* leaves not only the MU, but terminates the process) can occur in hard to predict ways (e. g., an exception falls through a `noexcept` function), we relax semantical correctness and allow that the output may not be fully actualized yet, which we deem an acceptable trade-off in the event of termination. Note, that memoization in this case is useless anyway, as the termination does not allow the same computation to reoccur during the execution of the program.

Finally, we deny the hardly used feature of non-local jumps (i. e., using `setjmp` and `longjmp`) inside the MU, since we introduce additional objects with non-trivial destructors, which would cause undefined behavior [9, 18.10§4] for many usages that would have been correct before memoization.

2.7 CV-Qualifiers

Our analysis of the cv-qualifiers `const` and `volatile` revealed: Pointer-to-`const`-parameters (see Sect. 2.5) allow us to deny their target to be output of a transparent function. Besides that, we do not rely on this qualifier as const-casts and the `mutable` keyword provide means to circumvent constness. The qualifier `volatile` guarantees that each implied memory access is actually performed in order. We argue that requesting to memoize a block containing `volatile` memory accesses is a fundamental contradiction.

2.8 Multi-Threading

Multi-threading is an optimization technique orthogonal to memoization. Our implementation assumes that only one thread is inside any MU at a time. Hence, we can demonstrate the feasibility on single-threaded programs. Future efforts clearly need to make the implementation thread-safe and investigate the impact on parallel programs.

3. AUTOMATED MEMOIZATION

In this section, we describe the design of our approach striving to provide automated memoization for pure or impure C++ code blocks using pointers in different ways, which can hence be used to avoid redundant computations in simulations. We specify the goals before we sketch the general approach and discuss the most important aspects in detail.

3.1 Design Goals

To maximize the benefit of our approach to automated memoization, we define the following three design goals:

Semantic Equivalence. Our approach converts an existing C++ code block into a memoized version of itself. It is the highest and most important goal that this transformation is sound, i. e., it does not change the results of the program. To this end, we define the transformed program to be *semantically equivalent* to the original if and only if it has the same semantics as defined by the C++ standard [9] except for: 1. a changed execution time and computational complexity, and 2. allocation and modification of additional memory (most notably the MC). Hence, we design our approach in a way that the perceivable effects and side effects of the original code and the memoized version are the same.

Maximize Memoizable Code. As discussed in Sect. 2, not every C++ statement can be memoized. For example, we cannot elude `volatile` reads without completely subverting the semantics of `volatile` qualification. Hence, we allow our approach to abort memoization if it would conflict with semantic equivalence, instead of issuing false results. However, we strive to maximize the number of supported C++ features. Our approach does support the basic C++ elements such as assignments, arithmetic operations, conditionals, loops, etc., as well as the challenging feature of input and output detection in the presence of pointer arithmetic. A detailed discussion of the actual capabilities and limitations follows in Sect. 3.6.

Optimize Efficiency. To maximize the benefit of the approach, its overhead should be as low as possible, such that its benefit can be reaped for computations of low computational complexity as well. To this end, the approach needs

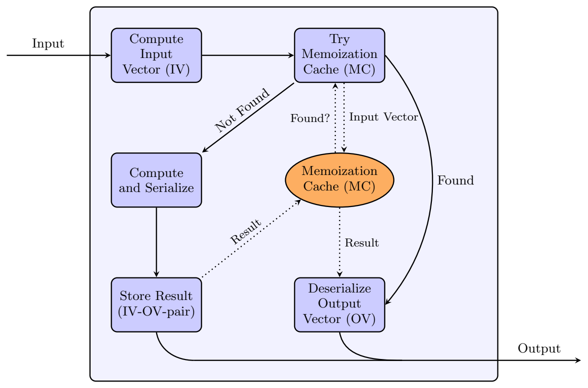

Input
Compute Input Vector (IV)
Try Memoization Cache (MC)
Not Found
Found?
Input Vector
Compute and Serialize
Memoization Cache (MC)
Found
Result
Result
Store Result (IV-OV-pair)
Deserialize Output Vector (OV)
Output

Figure 1: General memoization scheme.

```
void fib[[clang::transparent]](uint64_t* in) {
  // the anchor is already hardcoded
  // and does not need to be memoized
  if(*in <= 1) return;
  [[clang::memoize]] {
    // we can dereference or index pointers:
    auto in_minus_one = *in - 1;
    auto in_minus_two = in[0] - 2;
    fib(&in_minus_one);
    fib(&in_minus_two);
    *in = in_minus_one + in_minus_two;
  }
}
```

Figure 2: Recursive Fibonacci computation with pointer usage to illustrate automated memoization. The transformed code is listed in Fig. 3.

to minimize the data it considers input or output, i. e., overestimation should be avoided where possible. Additionally, the implementation can save overhead in three dimensions: 1. Each time a memoized code block is executed, the input has to be collected and the MC has to be queried for it. Minimizing this overhead improves performance of every execution of the memoized code. 2. When the result is found in the MC, it needs to be written to the corresponding output locations. Performing this as fast as possible maximizes the gain from using a cached result. 3. When the result is not found, it needs to be computed and stored in the cache. Not introducing too much overhead here is especially important for computations whose results can not be reused later on.

To demonstrate the feasibility of automated memoization we implemented it by code-to-code translation. A more efficient implementation could be realized by working at a lower level, though the implementation effort is much higher and we would not gain any additional scientific insights.

3.2 General Approach

Our approach to automated memoization works as follows. The developer annotates the MUs by adding the C++ attribute `[[clang::memoize]]`. This is the only manual action required; after handing the code to our tool, the memoization is performed automatically. A code-to-code translation rewrites each MU in a memoized way. Our proof-of-concept implementation utilizes a modified Clang to generate an abstract syntax tree as a basis for the memoization. This eases the implementation and allows debugging and verification of the generated, human-readable C++ code. For an efficient, product-level implementation we recommend integration into an actual optimizing compiler to further increase performance. However, this engineering effort is outside the scope of this paper. The transformed version performs the memoization as illustrated in Fig. 1. Fig. 2 lists the code of a recursive Fibonacci implementation using pointers to illustrate how memoization works with pointers. The transformed code is listed in Fig. 3, only slightly modified from the autogenerated code to fit in the paper: We renamed variables, applied indentation, changed line breaks, and removed parts not directly relevant to the memoization.

The automatic transformation identifies all read operations to exterior variables (cf. Sect. 2.2) and creates code that only reads these variables and serializes the input into the *Input Vector (IV)*. In the example, the only input is the object `in` points to, i. e., ("in",0). The IV is used as the key to search the entry in the MC. If it is not found, we execute a version of the original code that is slightly modified, such that output is not only written to its location in memory, but also serialized to the *Output Vector (OV)*. In the example, the only output is ("in",0) as well. The IV-OV-pair is then stored in the MC. On later execution of the same MU with the same input, the IV will be found in the MC, and the corresponding OV returned. Our memoized version of the code then skips the (expensive) computation, and directly deserializes the OV to the corresponding memory locations.

To this end, the values in the IV and OV are simply bitwise copies of the original values, regardless of their semantics. Hence, our approach needs no special handling for floating point numbers, but just performs bitwise comparisons.

To improve the performance of our transformation we require that interior or exterior objects be only accessed via paths originating from variables of the same category. Additionally, our implementation does not perform the memoization when the same object is accessed via multiple different paths. In the following we describe the procedure and the rationale for these decisions in more detail.

3.3 Input Vector Computation

We need to extract all read operations on exterior objects from the original code and create code that does as little as possible besides reading those values and storing them in the IV. It is of special importance to ensure that no writes to exterior objects are performed at this stage, as they might interfere with later stages. The straightforward approach would be to search the code for all reads of exterior objects. However, if a conditional occurs inside the code, we would not only overestimate the input, but also potentially crash the transformed program by dereferencing a null pointer that was originally protected by an `if`-clause.

Our algorithm uses two basic ideas to tackle the IV computation. First, instead of synthesizing completely new code to compute the IV, we slice the original code in such a way as to compute the IV without causing any other side effects and then remove as much of the code as possible. Second, we intercept not only reads but also writes which we can then store in a Temporary Cache (TC) instead of writing to the exterior object. Of course this means that we need to test the TC for every read as well to prevent stale reads.

The transformation begins by adding interceptions for all accesses of exterior objects. While such reads and writes can

```
#include <clang/memoize>
void fib[[]](::std::uint64_t *in) {
  // the anchor is already hardcoded and does not need to be memoized
  if (*in <= 1) return;
  /* transformed code begins here */ {
    try {
      auto __policy = ::std::clang::memoize::policy();
      static auto __dict = __policy.dict();
      auto __reader = __policy.reader();
      /* Read Key */ [&] {
        auto in_minus_one = __reader.read<1>(in, 0) - 1;
        auto in_minus_two = __reader.read<1>(in, 0) - 2;
      }();
      auto __iter = __dict.find(__reader.key);
      if (__iter != __dict.end()) {
        /* Check External Aliasing */
        for (auto const& __result : __iter->second.map) {
          switch (__result.first.baseid()) {
          case 1: __reader.alias(in, __result.first); break;
          }
        }
        /* Apply Result */
        for (auto const& __result : __iter->second.map) {
          switch (__result.first.baseid()) {
          case 1: __result.second.write_to(in, __result.first); break;
          }
        }
      } else {
        auto __results = __policy.results();
        auto __finalizer = ::std::clang::memoize::bits::at_scope_exit([&] {
          if(!__results.aliased()) __dict.emplace(::std::move(__reader.key), ::std::move(__results));
        });
        /* Compute Result */ {
          auto in_minus_one = *in - 1;
          auto in_minus_two = in[0] - 2;
          fib(&in_minus_one);
          fib(&in_minus_two);
          __results.write<1>(__reader, in, 0) = in_minus_one + in_minus_two;
        }
      }
    } catch (::std::clang::memoize::alias_exception const&) {
      auto in_minus_one = *in - 1;
      auto in_minus_two = in[0] - 2;
      fib(&in_minus_one);
      fib(&in_minus_two);
      *in = in_minus_one + in_minus_two;
    }
  } /* transformed code ends here */
}
```

Figure 3: Memoized version of the code in Fig. 2. This code has been automatically generated by our tool and then slightly edited for increased human readability and to fit in the paper format.

be caused in a multitude of ways (e. g., direct assignment, logical or arithmetic operation, passing arguments to functions), they can be reduced to the three basic cases of reads, writes, and reads followed by writes. For example, `a += b` performs a read on `b`, and both a read and a write on `a`.

To generate the IV, we simply store all reads on external objects that have not been read or written previously. Each value read is appended to the end of the IV to preserve the order. We can ignore repeated reads, as they cannot add any new information and we can ignore reads that follow writes to the same location, as the value that has been written is determined solely by reads that have been performed previously and thus been added to the IV already.

In the next section, we explain how our adapted dead code elimination allows the IV computation to be performed more efficiently while also explaining the need for a reliable alias detection. To explain how the TC creates an overlay address space and perform alias detection at the same time, we then further discuss its design and implementation.

Adapted Dead Code Elimination.

It is essential to avoid complex operations during IV reading. To this end, we use a transformation derived from standard dead code elimination, which we extend for this purpose. As opposed to simpler analyses this allows us to deal efficiently with complex IV calculations. One interesting class of cases in which this is especially important is that of reading zero-terminated arrays, as the last part of the IV may be read only very close to the end of the computation.

To this end, we apply a very broad definition of *dead code* in the attempt to create a program slice that is narrowly defined by its purpose to generate the IV. The basic premise of the proposed technique is to consider everything expendable but reads from exterior objects that have not been read from or written to before. Most importantly, this also includes writes to external objects that are never read afterwards. By applying common dead code elimination techniques, operations which are no longer necessary are successively removed. For example, if a value `x` is stored in an exterior variable, which is never read afterwards, we remove the write and subsequently all the code that computes `x` up to (but excluding) the point were its input was read.

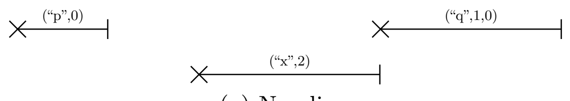
The effectiveness of this analysis depends on our ability to distinguish internal from external objects, which is problematic when considering not only scalar variables, but also pointers. Determining whether an object that is the result of a pointer expression is interior or exterior (as defined in Sect. 2.2) is not trivial. If the base pointer is interior, in most cases the final object will be interior as well. However, after creating a pointer locally, it might still be assigned the address of an exterior object. A similar problem occurs if an exterior pointer is assigned the address of an interior object.

In general, static code analysis does not allow to reliably deduce which object a pointer will point to in a given expression, as the decision whether it will point to an internal or an external object may depend on a runtime branch. A dynamic check could be performed, e. g., by determining whether the pointer points into the stack segment holding the local variables of the MU. However, the C++ standard does not guarantee the correctness of such an approach, instead it depends on the implementation of the compiler that later on translates the memoized version into executable code. Furthermore – and arguably more importantly – the runtime checks would introduce considerable overhead.

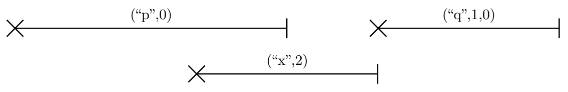
To be able to apply an efficient, standard-compliant solution, we restrict the usage of pointers inside an MU: An interior pointer must not store an address to an exterior object and vice versa. We discuss the impact of this decision in Sect. 3.6. Note that this property can easily be checked statically. This constraint allows us to determine whether an object reached by $(X, ...)$ references an interior or exterior object just by checking whether the symbol X itself is interior. Hence, we determine the input of an execution of the code block as a set $I = \{(X, ...)|X\text{read} \wedge X\text{not interior}\}$ and the output as $O = \{(X, ...)|X\text{written} \wedge X\text{not interior}\}$.

However, another requirement exists to ensure correctness: Since, in general, the same object may be reached via multiple different paths (as specified in Sect. 2.3), we had to assume that any write may change the result of any subsequent read, which would significantly inhibit the power of the dead code analysis. Instead, we perform the dead code elimination and read the IV as if aliasing would never happen. Although this tradeoff increases the potency of the dead code elimination, it also requires us to add another analysis that will ensure that no errors are introduced accidentally when attempting to memoize code that does indeed encounter aliased objects as discussed in the following.

In summary, our adapted dead code elimination leaves only the code to establish the IV as well as code that calculates what to include in the IV. It requires that objects are only ever accessed through a single path, a property that cannot be established at compile time. The TC discussed in the next section provides a way to detect aliasing and gracefully degrade to unmemoized execution in that case.



(a) No alias.



(b) Overlap with the previous object.

Figure 4: Alias detection: inserting ("x",2).

The Temporary Cache.

At the most basic level, the Temporary Cache (TC) is a dictionary that maps memory addresses[3] to paths, object values, and the length of objects. If a read or write causes a memory access that is not already in the TC, an entry is immediately inserted into the TC, which allows to satisfy all subsequent reads and writes. By using the TC to establish an overlay address space, all writes can be effectively prevented from being outwardly visible, while still being easily located when written objects are subsequently read again.

While, at first glance, the TC also seems to run into problems with aliasing, it is designed this way exactly to detect different paths leading to the same exterior object. Any possible alias falls into one of three categories: 1. The simplest case is that in which the alias is an exact match, as a simple lookup in the TC identifies the alias by comparing the stored path with the current one. 2. The current memory access begins in the range of a previously accessed object without matching exactly. To identify that case, it is only necessary to find the entry immediately preceding the target address, and check its end against the start of the current one. 3. The current memory access ends in the range of a previously accessed object without matching exactly. That is the case when the start address of the entry following the target address falls before the end address of the current memory access. A visualization of how the TC is used to detect aliases can be seen in Fig. 4, where the cross at the left hand of each element represents the base address and the line shows its size. Additionally, the TC contains a simple flag that tracks whether any alias has been found.

3.4 Performing Memoization

After computing the IV we need to determine whether the same input has occurred before. This operation is as trivial as searching for the IV in a hash map. On a hit, the MC returns the corresponding OV. Before actually applying the result, we need to finalize the aliasing check, which so far may miss locations that are only written (via two different paths). To this end we iterate over the OV and check the paths and locations found by tracing the path of each element to the actual memory location against the TC.

If the check shows that no aliases are encountered, the deserialization of the OV is performed by iterating over the OV again and storing the value in the object at that address. This effectively applies all side effects of the computation without having to execute the (complex) computation itself.

[3]We assume a flat memory model and that reinterpreting data pointers to `::std::uintptr_t` values has the obvious implementation, which is valid for the x86_64 platform and all our target compilers.

Should the IV have not been found in the MC, the OV is computed, which will also perform the original computation. The next section explains how this computation will lead to the correct result in both presence and absence of aliases.

3.5 Output Vector Computation

Computing the OV is considerably simpler than computing the IV, as all that needs to be done is to keep track of all writes being performed. Writes are tracked by storing tuples of paths and object values, which can then later be deserialized by following the paths from their respective roots. Memory writes are not completely intercepted during computation of the OV, but rather stored in their originally intended locations as well. The advantage of using paths versus storing the address is, amongst others, that it also works with dynamic variables whose actual address changes depending on the depth of the current function stack.

Since the computation of the IV is designed to eschew as many writes as possible, its alias detection cannot be complete. All missing alias checks are performed during the OV computation, which must necessarily perform all writes. Therefore, the combined alias detection is complete. Should no alias be detected, the OV is stored in the MC to be retrieved during future computations. It is not necessary to immediately deserialize the OV, as the output will already have been stored as a side effect of the OV computation. For the same reason, no further action is necessary if an alias has been detected at this stage, instead the memoization degrades gracefully, i. e., no entry is created in the MC.

3.6 Discussion

While our approach is the first viable development in automated memoization for decades (see Sect. 5), it still has rather relevant limitations. The scope of our work is to answer the research question whether automated memoization can be applied in the presence of non-trivial pointer expressions[4], i. e., those going beyond what is correctly memoized by treating pointers as ordinary numbers. We focus on automating the memoization process, not automatically deciding where memoization is applied for the biggest benefit.

With our approach many cases of trivial and non-trivial pointer usages can in fact be used inside the MU, with the only exceptions being those listed below:

- The lifetime of objects is coupled to the scope of the MU, i. e., objects created outside must not be destroyed inside and objects created inside must not persist the scope of the MU[5]. In other words, the effects of creating or deleting objects cannot be stored in the OV, which only stores modifications of existing objects. The former can be added to our implementation, by allowing the OV to hold a representation of those effects and applying them – i. e., creating or deleting the corresponding objects – during OV deserialization.
- Addresses of exterior objects must not be stored in interior objects and addresses of interior objects must not be stored in exterior objects. We argue that the former can easily be rewritten by not creating an interior copy of the pointer, but using the exterior variable, potentially with an interior index, instead. The latter form of pointer usage is never necessary as it does not provide any benefit if the pointer's lifetime is greater than the lifetime of the object it points to. Hence, an interior pointer could be used instead. A violation of either rule is easily discovered at compile time by also disallowing such assignments in unreachable code.
- As a result of this work, we found that aliasing is a severe problem to automated memoization. Assume two pointers `p` and `q` point to the same object o, and the object is modified via `p` before it is read via `q`. In this case the initial value of o is not read and hence not relevant to the computation. If, however, in the next execution of the MU, `p` and `q` point to different objects, the initial value of the object `q` points to is in fact read. Hence, the memoized code had to handle the two cases differently. Since aliasing cannot be generally checked at compile time [12], an implementation cannot be able to generate code for each possible case. To this end, we conclude that automated memoization with fully unrestricted pointer usage is infeasible as the runtime overhead associated with a dynamic approach would be overwhelming. We decided to perform a much simpler *dynamic* aliasing analysis (see Sect. 3.3), which detects aliases, but needs to abort the memoization in such a case. This enables applicability of memoization if no aliasing occurs, ensures correct results in either case by gracefully degrading to unmemoized execution, and can issue a warning to enable the developer to potentially resolve the aliasing by using only one pointer to access each object inside the MU.

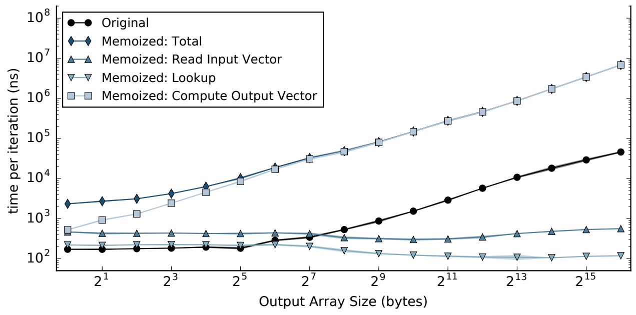
Additionally, an obvious limitation is that we cannot analyze code that is not available to the memoizer (e. g., precompiled libraries). Hence, if function calls appear in the MU, the memoizer requires additional information. If the requirements for inlining the function call are fulfilled, the analysis can continue inside the function implementation. If, however, it cannot be statically determined which function is called (e. g., due to an untraceable function pointer), the implementation is not available to the memoizer (e. g., precompiled), or the number of function calls cannot be predicted (e. g., recursion), the user has to provide a transparency annotation or the transformation needs to be aborted with a warning. This poses an additional requirement to the user over the actual intent to only request for a single annotation to memoize a block. However, this cannot be avoided as unknown code cannot be analyzed. We argue that especially for libraries commonly used to perform expensive computations (e. g., mathematical functions), this could already be performed by the library maintainer. As an additional bonus, a manual annotation allows us to annotate functions as transparent that are not pure in the strictest sense[6].

Finally, certain features like multi-threading, which are not in the scope of answering the research question stated above, are not in scope of this work.

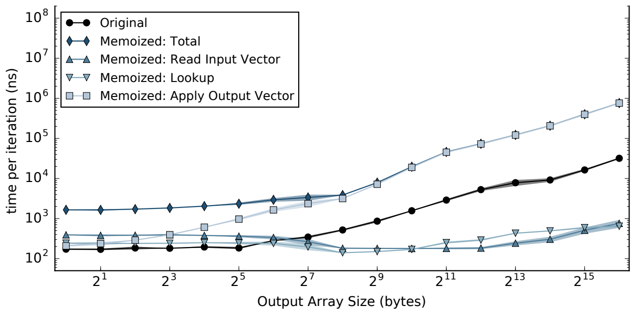
[4]And references, treated very similarly, see Sect. 2.4.

[5]Function-level static variables are not created during the first execution of the MU, but only initialized then. To ensure that this is performed correctly, they receive additional treatment – basically a simple flag – to ensure that they are initialized only once and the initialization expression only contributes to the IV once in the program execution.

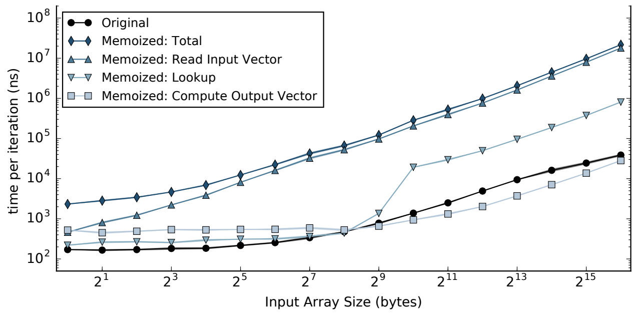
[6]For example, we did encounter functions using variables with static storage duration as scratch space, instead of variables with automatic storage duration. Annotating that function as transparent is in line with our definition, as the changes to that scratch pad are irrelevant side effects – they are overwritten every time the function is called.



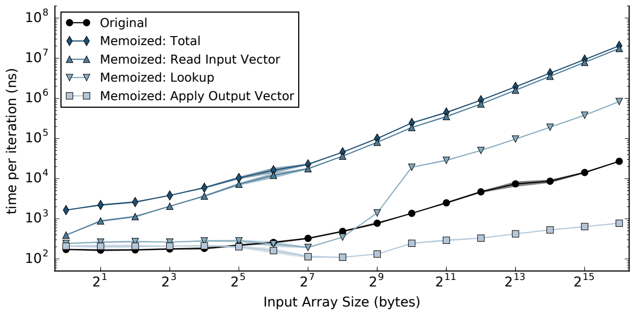
(a) $N_\mathrm{I} = 1$ (varying OV size), 1. iteration of outer loop.



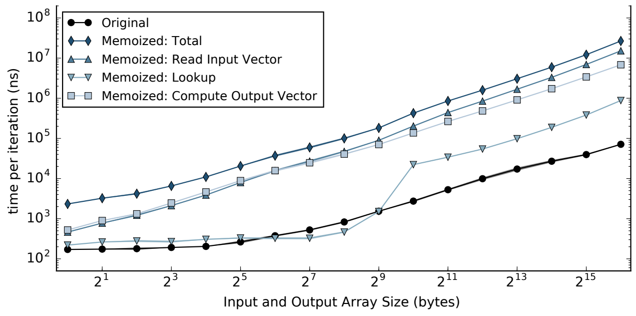
(b) $N_\mathrm{I} = 1$ (varying OV size), 2. iteration of outer loop.



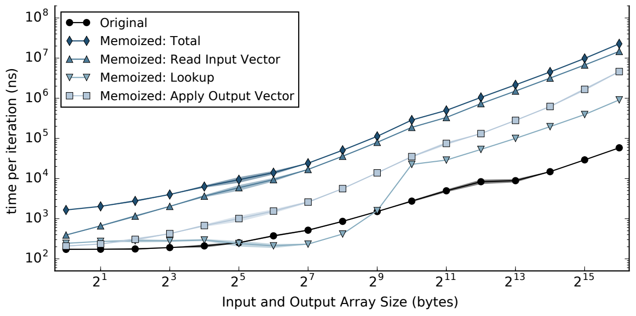
(c) $N_\mathrm{O} = 1$ (varying IV size), 1. iteration of outer loop.



(d) $N_\mathrm{O} = 1$ (varying IV size), 2. iteration of outer loop.



(e) $N_\mathrm{I} = N_\mathrm{O}$ (IV, OV varying), 1. iteration of outer loop.



(f) $N_\mathrm{I} = N_\mathrm{O}$ (IV, OV varying), 2. iteration of outer loop.

Figure 5: Overhead evaluation: synthetic benchmark where memoization cannot gain benefit. Used to measure the overhead.

In summary, we found that pointers do not generally prevent automated memoization. If the above mentioned restrictions are met, a large and useful subset of pointer expressions can in fact be memoized in an automated fashion. We observed that especially complex computations whose memoization seems promising often use pointers to access (jagged multi-dimensional) arrays. Our approach of identifying objects via paths in fact allows this usage of pointers and enables us to determine the actual input and output of the MU. Hence, we conclude that we are able to apply automated memoization to programs using pointers in a multitude of ways, while certain restrictions have to be fulfilled to avoid undecidable problems.

4. EVALUATION

We evaluate our approach by first performing overhead measurements to derive simple formulae to give a basic estimate when our approach is beneficial. These approximations can assist model developers in selecting appropriate regions for memoization and can as well be used in future research on automatic identification of such regions. Second, we implement the well-known Fibonacci-computation in a recursive implementation gratuitously using pointers to show the feasibility of our approach on recursive and pointer-based code. Finally, a case study performing a parameter study of an OFDM (orthogonal frequency-division multiplexing) network simulated by OMNeT++ [22] demonstrates the practical applicability of our approach in the simulation context.

All measurements are performed on a Xeon E5 compute server with 32 GB of RAM. Each experiment is repeated at least 5 times, all plots depict the means and 99 % confidence intervals. The latter are hard to perceive in many cases due to the low variance of the results.

As the optimization only makes sense if the quickly computed results are as well correct, we validated the output.

In each experiment we compared the computational results of the memoized version against the results of the original implementation. The results were identical in every case. Hence, we conclude that for every experiment performed in this chapter the transformation was performed correctly and maintained the semantics of the original program.

4.1 Overhead Evaluation

We measure the overhead by means of a simple synthetic benchmark: An array of N_I 8-bit numbers is read, the numbers are aggregated to a 64 bit variable, which is then xor-folded to yield an 8-bit result. An array of N_O 8-bit numbers is filled with numbers calculated by adding the array index of the respective element to the above mentioned result. Since the computational effort is almost negligible, the memoization cannot speed up the computation, instead allowing the memoization overhead to be observed. Varying N_I and N_O directly varies the size of the IV and OV, which are the primary influence factors for the memoization costs.

The surrounding evaluation program consists of two nested loops. The inner loop is repeated 250 times, each time with a different input. The outer loop is executed twice, such that the inner loop is executed again for each of the 250 inputs used in the first iteration. Hence, the memoized code adds 250 items to the MC in the first outer loop iteration while none of the lookups is successful. In the second iteration, each result is retrieved from the MC. In the original code, both iterations behave exactly identical.

We performed experiments while growing only N_I, only N_O, and growing both simultaneously. Fig. 5 depicts the average runtime per inner loop iteration for each of the two outer loop iterations. The runtime of the memoized version is decomposed into the components of the memoization (IV computation and MC lookup for both iterations, OV computation or application for the 1. or 2. iteration, respectively).

We observe that the MC lookup only contributes very little to the total runtime (note the logarithmic scale). For the first iteration we observe that for large output arrays the overall runtime is almost equivalent to the runtime of the OV computation whose cost primarily depends on N_O (at about $150\,\text{ns} \cdot N_\text{O}$). For smaller output arrays additionally the IV reading becomes relevant, which costs about $300\,\text{ns} \cdot N_\text{I}$. As a very rough approximation we conclude that the memoization overhead ranges about $T_\text{F} = 150\,\text{ns} \cdot N_\text{O} + 300\,\text{ns} \cdot N_\text{I}$ if the lookup is not successful. However, this simple approximation ignores, for example, the effect that increasing the input size as well influences the OV computation due to non-uniform memory access and growing data structures.

Similarly, we analyze the second iteration of the outer loop: The costs of computing the IV are the same as above, as the computation has to be performed in both cases, i. e., we observe overhead of about $300\,\text{ns} \cdot N_\text{I}$. For the application of the OV we observe about $15\,\text{ns} \cdot N_\text{O}$. We conclude that we can approximate the overhead for a successful lookup by $T_\text{S} = 15\,\text{ns} \cdot N_\text{O} + 300\,\text{ns} \cdot N_\text{I}$. Furthermore, we observed (without figure) memory overhead linear in both N_I and N_O.

From these approximations we derive that our approach to automated memoization pays off if sufficient memory is available to hold the MC and $p \cdot T_\text{S} + (1 - p) \cdot (T_\text{F} + T_\text{C}) < T_\text{C}$ with the computational costs of the original MU of T_C and the fraction p of calls that can be served from the MC. Obviously, the bigger p and T_C the bigger the gain.

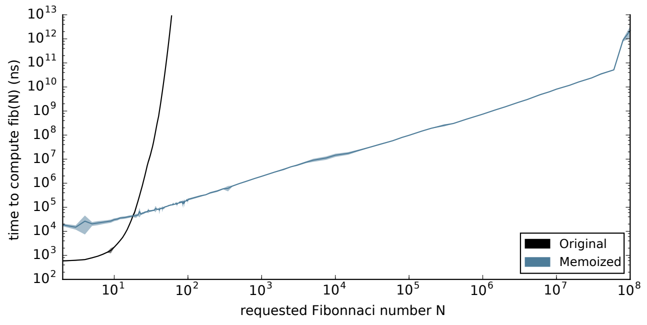


Figure 6: Performance of recursive Fibonacci computation using pointers.

4.2 Recursive Fibonacci with Pointers

To demonstrate the potential benefit of memoization and the versatility of our approach, we implemented a naïve recursive function (see Fig. 2) that gets an integer n and computes the nth Fibonacci Number ($F_n := F_{n-1} + F_{n-2}$, $F_0 := 0$ and $F_1 := 1$), or more precisely the bits of F_n that fit into an integer, i. e., $F_n \bmod 2^{64}$. To demonstrate the viability of our approach for impure code, the function takes a pointer to an integer that contains the actual input and stores the result in that same object. However, the function is transparent (see Sect. 2.5) and annotated accordingly to allow the memoization of this recursive function. It must be noted that, should the IV generation not be optimized well enough (cf. Sect. 3.3), this function would perform the whole computation before any memoization takes place.

Fig. 6 clearly shows the expected exponential runtime of the unmemoized algorithm (computing the nth Fibonacci Number recursively with this algorithm takes $\Theta(\varphi^n)$ time, with $\varphi \approx 1.618$ being the golden ratio), and that the memoized version also performs as expected by only requiring linear time up to $F_{6\cdot10^7}$. For $F_{8\cdot10^7}$ the compute server's memory doesn't suffice to store the MC and we observe a slowdown caused by swapping. The overhead of the memoization is also clearly visible for small n, where the unmemoized code is significantly faster. Since the axes are both scaled logarithmically, it is easy to overestimate the actual difference for small n – the memoized code runs in only a few microseconds. Starting at $n \approx 18$, the unmemoized code requires more time than the memoized code. This also means that a small table assist of 17 values would cause the memoized code to always outperform the unmemoized code.

4.3 Case Study: Network Simulation

To demonstrate the feasibility of our approach in a practical use case, we apply it to a parameter study of wireless network simulation. The simulation model is implemented for the open source simulation framework OMNeT++ [22] in C++. In the simulation model, a set of wireless nodes is placed on a 1 by 1 km area. A number of those nodes transmits frames in a fixed pattern. A channel model based on the Friis path loss model [3] and a complex OFDM fading model [23] calculates the received signal strength. In the parameter study the total number of wireless nodes is varied in 14 steps from 1 to 50, the fraction of transmitting nodes in 9 steps from 1 % to 100 %, and the time between two transmissions in 5 steps from 1 µs to 10 ms. Each experiment is

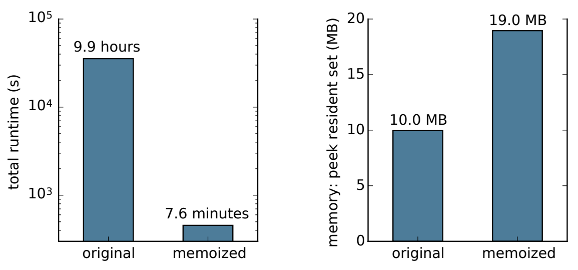


Figure 7: Case study: simulation parameter study.

repeated 10 times with different random number generator seeds, hence the total parameter study consists of 6300 runs.

We identified the fading computation as a good candidate for memoization as it is a complex operation, its input is small and repeated frequently. This fading computation heavily uses pointers to iterate over multi-dimensional arrays. Existing approaches to automated memoization would treat these pointers by their address, hence compute false results. However, as for the other experiments we compared the computational results to those of the original implementation and observed exactly the same results. As this fading computation seemed most promising, we only tagged this block for memoization. We executed the parameter study in the original as well as the memoized version, both with OMNeT++ 5.0b3 on the above described hardware.

The results of our experiments are depicted in Fig. 7. In the original implementation, each run took about 5-6 s, resulting in a total runtime of about 10 hours. In the memoized version, we observed a similar runtime for the first run, where no computations could be omitted. However, from the second run on, we observed significant speedups, certain runs were completed in as little as 5 ms. Completing the total parameter study then took less than 8 minutes, hence our automated memoization yielded a speedup of about 75×.

The MC, on the other hand, doubled the memory consumption of the program. We feel confident asserting that a penalty of less than 10 MB will be happily accepted by a user who now only has to wait minutes instead of hours.

5. RELATED WORK

Memoization was first introduced by Michie [15, 16] and implemented in a framework by Popplestone [19] in 1967. Though the framework provides an interface and assists the user, the challenging parts have to be realized completely manually. In particular, the user needs to implement a function deciding whether two inputs are equal, i. e., the user has to determine the input. Mostow and Cohen [17] provide an in-depth analysis of the memoization idea and the resulting challenges like side effects. They propose to display a list of side effects to the user and ask for permission to memoize, ignoring the side effects. This might be possible in certain cases, however, recognizing and applying side effects correctly makes our approach by far more generally applicable.

Several approaches realizing automated memoization have been implemented, for example those by Norvig [18], Hall et al. [4, 13, 14], and Hinze [6]. These approaches use Haskell and Lisp, but also C++ as the basis of their implementation. In the functional language Haskell every function is by definition pure, hence input and output is given by the function definition. Though Lisp supports imperative programming with functions modifying global state, i. e., inducing side effects, the approaches explicitly restrict themselves to pure functions. This also holds for the C++ implementation [14], which effectively adopts the Lisp approach from [13] to C++. Hence, input may only be provided in function parameters, only the return value may be output, and pointers are just handled like integers, i. e., pointers to the same address are treated equally even if the value at that address has changed, which inevitably introduces errors if the pointed to object is actually input. Similarly, in logic programming languages the concept of tabling is used to memoize results of previously evaluated (by definition pure) rules [24]. To the best of our knowledge no generic approach to automated memoization without restriction to pure functions has been proposed to date.

Tsumura et al. [10, 21] propose to integrate memoization functionality directly into the processor hardware. While this is probably the most promising approach to speed up any software independent of the programming language and paradigm, the proposed hardware is not available to most users, i. e., software implementations are essential for wide applicability. To this end, we provide an approach implemented in software and able to cope with impure code in impure languages to speed up simulations in practice.

To avoid unnecessary computations in simulation parameter studies simulation cloning [7, 8] and updateable simulations [2] should be mentioned. Both techniques share the motivation of our approach. However, simulation cloning clones any affected “virtual logical process” of the simulation as soon as the state of that process deviates. Hence, later occurring re-computations, which base only on parts of the state of that process, can not be avoided. Updateable simulations can avoid a large set of re-computations. However, the major limitation of that approach is the requirement to implement update functions realizing the necessary functionality to compute the differences between two runs. Like manual memoization this is a labor-intense, error-prone effort that needs to be carried out by the model developer.

6. CONCLUSION

In order to avoid redundant re-computations of intermediate results in simulation parameter studies by means of memoization, two major steps have to be approached. First, promising code blocks have to be identified whose effort can be saved using memoization. Second, the code blocks have to be rewritten in a way that the results are cached and can be retrieved from that cache instead of re-computation.

In this paper, we focus on automating the second step of this procedure and describe our approach for impure languages, realized in a proof-of-concept implementation for C++. While up to now every existing technique requires a function to be pure in order to be able to apply automated memoization, our approach detects the full input and output even if accessed via pointers, and hence eliminates this restriction. However, as discussed in Sect. 3.6, certain restrictions like absence of local aliasing are still required to be enforced as to avoid undecidable problems.

Once a developer has identified a suitable code block for memoization, the block can be annotated using a single attribute. Our tool then parses the code and generates a memoized version after detecting the full input and output. Hence, the approach is viable for both pure and impure computations even when using pointers in several ways.

Our evaluation shows the practical feasibility of the approach. In general, the approach is promising if the memoized computation is complex enough and executed several times on the same input. In Sect. 4.1 we derive simple approximations to estimate under which conditions the approach is promising. In a simulation parameter study we observed a 75× speedup while only increasing memory consumption by 9 MB. We conclude that automated memoization can significantly help reducing the time developers have to wait for their results with minimal manual effort.

Future efforts should address the automatic identification of promising computations, such that annotation by the user is no longer required. Additionally, instead of always adding an entry to the MC, selection strategies could be developed to reduce overhead and memory consumption if the result of a computation can be expected to not being reused later. Hence, the execution could switch between memoized and unmemoized versions of the code. Furthermore, our approach needs to cope with multi-threaded software, allowing multiple threads to concurrently and cooperatively utilize a common MC. This allows to combine the power of both PDES and memoization to benefit from both. Finally, the performance of the proof-of-concept implementation can be improved to reduce the overhead and make memoization promising for computations of less complexity. Nevertheless, our approach already demonstrates the feasibility of automated memoization for impure languages as used by many simulation tools and yields promising speedups.

Acknowledgments

The research leading to these results has received funding from the German Research Foundation (DFG) under Agreement n. 625799 (MemoSim) and from the European Research Council under the EU's Horizon2020 Framework Programme / ERC Grant Agreement n. 647295 (SYMBIOSYS).

7. REFERENCES

[1] E. W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[2] S. Ferenci, R. Fujimoto, M. Ammar, K. Perumalla, and G. Riley. Updateable Simulation of Communication Networks. In *Proc. of the 16th Workshop on Parallel and Distributed Simul.*, pages 107–114, 2002.

[3] H. T. Friis. A Note on a Simple Transmission Formula. *Proc. of the Institute of Radio Engineers*, 34(5):254–256, 1946.

[4] M. Hall and J. Mayfield. Improving the Performance of AI Software: Payoffs and Pitfalls in Using Automatic Memoization. In *Proc. of the 6th Intl. Symposium on Artificial Intelligence*, 1993.

[5] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 Project Goals. In *Proc. of the 1st Workshop on ns-2: the IP network simulator*, 2006.

[6] R. Hinze. Memo Functions, Polytypically! In *Proc. of the 2nd Workshop on Generic Programming*, pages 17–32, 2000.

[7] M. Hybinette and R. Fujimoto. Cloning: A Novel Method for Interactive Parallel Simulation. In *Proc. of the 29th Winter Simul. Conf.*, pages 444–451, 1997.

[8] M. Hybinette and R. M. Fujimoto. Cloning Parallel Simulations. *ACM Transaction on Modeling and Computer Simul.*, 11(4):378–407, 2001.

[9] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Dec. 2014.

[10] K. Kamimura, R. Oda, T. Yamada, T. Tsumura, H. Matsuo, and Y. Nakashima. A Speed-up Technique for an Auto-Memoization Processor by Reusing Partial Results of Instruction Regions. In *Proc. of the 3rd Intl. Conf. on Networking and Computing*, pages 49–57, 2012.

[11] D. E. Knuth. Structured Programming with go to Statements. *ACM Comp. Surveys*, 6(4):261–301, 1974.

[12] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[13] J. Mayfield, T. Finin, and M. Hall. Using Automatic Memoization as a Software Engineering Tool in Real-World AI Systems. In *Proc. of the 11th Conf. on Artificial Intelligence for Applications*, pages 87–93, 1995.

[14] P. McNamee and M. Hall. Developing a Tool for Memoizing Functions in C++. *ACM SIGPLAN Notices*, 33(8):17–22, 1998.

[15] D. Michie. Memo functions: a language feature with "rote-learning" properties. Technical report, Edinburgh University, Dept. of Machine Intelligence and Perception, 1967.

[16] D. Michie. Memo Functions and Machine Learning. *Nature*, 218(5136):19–22, 1968.

[17] J. Mostow and D. Cohen. Automating Program Speedup by Deciding What to Cache. In *Proc. of the 9th Intl. Joint Conf. on Artificial Intelligence*, pages 165–172, 1985.

[18] P. Norvig. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[19] R. Popplestone. Memo functions and the POP-2 language. Technical report, Edinburgh University, Dept. of Machine Intelligence and Perception, 1967.

[20] D. Tsafrir, R. Wisniewski, D. Bacon, and B. Stroustrup. Minimizing Dependencies within Generic Classes for Faster and Smaller Programs. *ACM SIGPLAN Notices*, 44(10):425–444, 2009.

[21] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima. Design and Evaluation of an Auto-Memoization Processor. In *Proc. of the 25th Intl. Multi-Conf. on Parallel and Distributed Computing and Networks*, pages 230–235, 2007.

[22] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proc. of the 15th European Simul. Multiconference*, 2001.

[23] C. Wang, M. Pätzold, and Q. Yao. Stochastic Modeling and Simulation of Frequency-Correlated Wideband Fading Channels. *IEEE Transaction on Vehicular Technology*, 56(3):1050–1063, 2007.

[24] N.-F. Zhou and T. Sato. Efficient Fixpoint Computation in Linear Tabling. In *Proc. of the 5th ACM SIGPLAN Intl. Conf. on Principles and Practice of Declarative Programming*, pages 275–283, 2003.