On Automated Memoization in the Field of Simulation Parameter Studies

MIRKO STOFFERS, DANIEL SCHEMMEL, OSCAR SORIA DUSTMANN, and KLAUS WEHRLE, Communication and Distributed Systems, RWTH Aachen University

Processes in computer simulations tend to be highly repetitive. In particular, parameter studies further exasperate the situation as the same model is repeatedly executed with only partially varying parameters. Consequently, computer simulations perform identical computations, with identical code, identical input, and hence identical output. These redundant computations waste significant amounts of time and energy.

Memoization, dating back to 1968, enables the caching of such identical intermediate results, thereby significantly speeding up those computations. However, until now, automated approaches were limited to pure functions. At ACM SIGSIM-PADS 2016 we published, to the best of our knowledge, the first practical approach for automated memoization for impure code. In this work, we extend this approach and evaluate the performance characteristics of a number of extensions that deal with questions posed at PADS: (1) To reduce and bound the memory footprint, we investigate several cache eviction strategies. (2) We allow the original and the memoized code to coexist via a runtime-switch and analyze the crossover point, thereby mitigating memoization overhead. (3) By optionally persisting the Memoization Cache to disk, we expand the scope to exploratory parameter studies where cached results can now be reused across multiple simulation runs.

Altogether, automated memoization for impure code is a valuable technique, the versatility of which we explore further in this article. It sped up a case study of an OFDM network simulation by a factor of more than 80 with an only marginal increase of memory consumption.

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

Additional Key Words and Phrases: Automatic memoization, accelerating parameter studies, impure languages

ACM Reference format:
Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2018. On Automated Memoization in the Field of Simulation Parameter Studies. *ACM Trans. Model. Comput. Simul.* 28, 4, Article 26 (September 2018), 25 pages.
https://doi.org/10.1145/3186316

This research was funded by the German Research Foundation (DFG) under contract number 625799. The research leading to these results has received funding from the European Research Council under the EU's Horizon2020 Framework Programme/ERC Grant Agreement number 647295 (SYMBIOSYS).
Authors' addresses: M. Stoffers, D. Schemmel, O. Soria Dustmann, and K. Wehrle, Communication and Distributed Systems, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany; emails: {stoffers, schemmel, dustmann, wehrle}@comsys.rwth-aachen.de.

1049-3301/2018/09-ART26 $15.00
https://doi.org/10.1145/3186316

1 INTRODUCTION

Computer simulations are programs with highly repetitive computations in which the same event handlers are executed repeatedly, often on the same input as before. In parameter studies, this effect is amplified further by executing multiple configurations and executing each configuration multiple times with different random number seeds. While the seeds naturally need to be different for each execution of any individual configuration, the repetitiousness of the whole process is increased even more when the seeds are the same for different configurations. We conclude that, overall, 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 in 1968 [20]. Michie developed the idea of so-called memo functions (now known as *memoized functions*) that reuse previously computed results. 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, this article focuses on the second step. We discuss an approach to tackle the first step in Reference [28]. Alternatively, model developers can annotate the promising code blocks and save the effort of manually applying the memoization transformation. The actual memoization is then performed by an automated tool. Unfortunately, all previously developed techniques to automated memoization are designed for pure functions[1] only. However, in most simulation models the computations heavily rely on impure code. Hence, techniques restricted to pure functions can not be applied to a large set of simulation models.

An approach to *automated memoization for impure code* can, however, 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. In Reference [29], we first introduced our approach to automated memoization, which does not rely on the purity of the code to be memoized. Our implementation operates on C++ [12], which is used to implement models for the popular open source simulation frameworks ns-3 [8] and OMNeT++ [31], 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 our work that it is necessary to pose certain restrictions on pointer usage, for example, to avoid undecidable problems such as static aliasing analysis [14].

To apply the automated memoization, we parse the provided C++ code using Clang,[2] identify all input and output, and generate new C++ code with a memoized version of the original. 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

[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 the MC. Hence, our approach works on pure *and* impure computations. A proof-of-concept implementation is available online.[3]

This article builds on Reference [29] and additionally contributes the following enhancements: (1) We apply common cache eviction strategies on the MC to reduce and bound the memory needed by automated memoization. (2) To save even more memory and the time required to perform the key derivation in cases where memoization cannot yield a benefit, we enable the developer to choose whether or not to use memoization on a case-by-case basis. (3) To decrease the time until the MC yields hits, we add an option to persist the MC to permanent storage. This is particularly beneficial for exploratory parameter studies where each simulation run needs to be a separate process and could therefore not reuse results from a previous run.

We evaluate the basic idea as well as the enhancements on synthetic benchmarks to characterize overhead and potential gain. We furthermore conduct experiments with a network simulation case study as well as random number generation to investigate the practical suitability in the area of modeling and simulation.

The remainder of this article is structured as follows. We first analyze the problem and its challenges more thoroughly (Section 2). After that, we introduce the design of our approach (Section 3). We demonstrate the practical feasibility by evaluation results (Section 4). We then discuss related work (Section 5) before we conclude the article (Section 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 semantic meaning to the computation (e.g., a pointer that is being dereferenced to access another object).

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

2.1 Memoization Granularity

All previous 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 permit side effects, we need to analyze the actual implementation anyway and the restriction to functions is no longer useful. For this reason, we enable 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, then it is highly unlikely that the results can be reused as the input of both computations must have occurred before in that combination. However, if the memoization unit ends before a computation is complete, then several intermediate

[3] https://code.comsys.rwth-aachen.de/projects/memoize/

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 C++ Language Constructs

To investigate how a memoized alternative can be created from non-memoized C++ code, we need to analyze the source language. We provide a detailed analysis in Reference [29]. Important to note is that to determine input and output of a computation, variables used during the computation need to be classified by their scope. We define a variable *interior* if it is local to the MU and has automatic storage duration. All other variables are *exterior*. Changes to exterior objects can be caused and/or perceived from outside the MU and must hence be considered input and/or output of the computation if read and/or modified, respectively.

To correctly reflect the semantics of pointers, we use a path notation to describe an object by its path through the object graph. For example, `*p=42` modifies the object at ("p",0). This also enables representing arbitrarily complex pointer expressions such as `(*(p+4))[3]=q[8][-5][3]`: The object at ("q",8,-5,3) is read, the object at ("p",4,3) is altered. Indirect addressing, such as `a[x]`, is also covered, since the value of `x` is first read, which yields a concrete value that can be used in the path ("a",*x*).

For function calls we need to differentiate between calls to functions whose code is known at compile time and calls for which this is not the case. For the latter we need to assume unknown side effects (such that memoization is not possible) or rely on an annotation signaling that it is safe to memoize. We allow the user to annotate functions or function calls as *transparent*. 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`), then 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. As input we treat the parameters' values or, for arguments of pointer or reference type, the object they point to.

Other side effects, such as terminal output, disk I/O, or memory allocation and deallocation are not supported by our implementation as long as they cannot be neglected and annotated as transparent. However, these effects could be covered in future implementations by adding the opportunity to store actions like "allocate 5 bytes of memory" into the output vector.

2.3 Multi-Threading

Multi-threading is an optimization technique orthogonal to memoization. Our implementation assumes only one thread is inside any MU at a time and no other thread relies on the order in which this thread accesses shared data. This does not mean multi-threaded programs are fully incompatible with the current implementation. The user can, for example, put the MU into a critical region, such that a lock is acquired before the MU can be reached, and hence only one thread is in the MU at a time. Other threads accessing the same data items and relying on the order in which the data items are accessed by other threads should acquire the same lock before entering such regions. In this context, it must be noted that, as long as no explicit counter-measures are taken, reordering as a compiler optimization is not that restricted anyway. To conclude, multi-threading can be used in a limited way when special care is taken.

Additionally, parallelism can be exploited by running multiple independent processes in parallel. This lets each process apply memoization independently of the other processes. If—as is often

the case when running large parameter studies—more processes need to be executed than CPUs are available, then the MC can as well be shared between subsequently executed processes by persisting it to disk. In this case, of course, it is again necessary to avoid data races when writing to disk.

At this point, parallelizing memoized programs is possible to a limited extent, but the challenges need to be analyzed in more detail before they can be solved in a broader scope. For the purpose of demonstrating the feasibility of automated memoization in the first place, we assume that the input program is single-threaded in this article.

2.4 Runtime and Memory Tradeoffs

The major goal of memoization is reducing the time it takes to compute the results of the program being executed. To this end, on the first occurrence of a certain input an entry in the MC is created. This causes additional processing overhead and increases memory consumption. To reap any rewards, the same input needs to be given at least once more, at which point the request may be satisfied from the MC.

Hence, we face two tradeoffs: First, the initial iteration incurs an additional processing overhead, which might or might not pay off eventually. Second, even when overall beneficial, memoization always trades memory consumption for processing time.

To address the first tradeoff, we provide an opportunity to decide at runtime whether to take the memoization branch or bypass it to compute the result in the original time and memory complexity. We need to emphasize that this decision poses a complex problem by itself, which we do not address in this article. Instead, we provide the developers with the opportunity to decide themselves which branch to take.

The time-memory tradeoff becomes an issue when limits on memory size are reached or approached close enough such that other optimizations that face time-memory tradeoffs as well are no longer performed (e.g., memory allocation algorithms that take more time to find available memory chunks under memory pressure). Due to the fairly large size of memory available in today's computers this is not an issue in many cases. Nevertheless, this problem needs to be addressed, since the MC has the potential to fully fill up all memory. Hence, we show how to specify an upper limit for the size of the MC and explore several strategies to evict entries when that limit is reached. To this end, we demonstrate MC implementations that provide common cache eviction strategies to keep the number of MC entries limited. Also, the aforementioned opportunity to decide whether to perform memoization in the first place can be used in this context to avoid filling the MC with rarely needed entries or to avoid thrashing.

2.5 Storage of Memoization Cache

To enable fast access to the MC, it must reside in main memory. However, in the context of modeling and simulation, memoization is particularly beneficial for parameter studies, which are commonly conducted in one of two ways: (1) All configurations are specified beforehand and a single process is spawned that performs the complete parameter study. (2) An individual process is spawned for each configuration. One example where this is necessary are exploratory parameter studies, where users may evaluate previous results before defining and executing additional configurations. To support both approaches, we provide an easy way of serializing and deserializing the MC to and from permanent storage so that it can be reused by subsequent processes.

3 AUTOMATED MEMOIZATION

In this section, we describe the design of our approach striving to provide automated memoization for pure and impure C++ code blocks using pointers in different ways to be used for avoiding

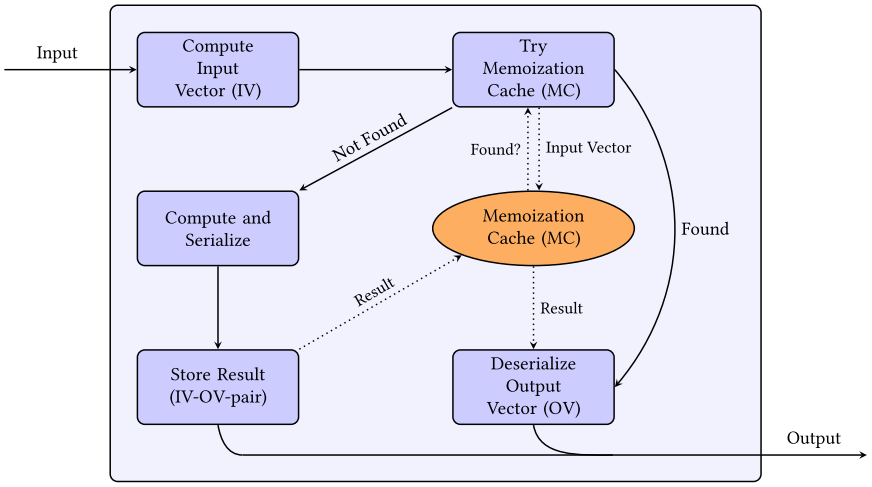


Fig. 1. General memoization scheme.

redundant computations. We specify the goals and outline the general approach before discussing the most important aspects in more detail.

3.1 Design Goals

As discussed in more detail in Reference [29], our approach targets three goals: First, any memoization tool useful to model developers must ensure semantic equivalence. This means that the perceivable effects and side effects of the original code and the memoized version are the same. Second, we strive to maximize the number of supported C++ features to enable memoizing as much code as possible. If any features that are not supported by the approach (such as certain kinds of aliasing) are encountered, then we always ensure semantic equivalence by aborting the memoization and simply executing the original code. Finally, we strive to generate code that runs as fast and with as little memory as possible without violating either of the other goals.

3.2 General Approach

To use our approach to automated memoization, the developer annotates the MUs. This is the only manual action required and can be automated by other approaches such as the one we discuss in Reference [28]. In either case, the approach discussed here then performs the memoization 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, debugging, and verification of the generated, human-readable C++ code. For an efficient, production-grade implementation, we recommend integration into an actual optimizing compiler to further increase performance. However, this engineering effort is outside the scope of this article. The transformed version performs the memoization as illustrated in Figure 1. Figure 2 lists the code of a recursive Fibonacci implementation using pointers to demonstrate how our tool handles pointers. The transformed code is listed in Figure 3, only slightly modified from the auto-generated code to fit in the article: 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. Section 2.2) and creates code that only reads these variables 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

```
1 void fib[[transparent]](uint64_t* in) {
2   // the anchor is already hardcoded and need not be memoized
3   if(*in <= 1) return;
4   [[memoize]] {
5     // we can dereference or index pointers:
6     auto in_minus_one = *in - 1;
7     auto in_minus_two = in[0] - 2;
8     fib(&in_minus_one);
9     fib(&in_minus_two);
10    *in = in_minus_one + in_minus_two;
11  }
12 }
```

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

MC. If it is not found, then 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.

The values in the IV and OV are 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 interior or exterior objects be only accessed via paths originating from variables of the same category. Our implementation does not perform the memoization when the same object is accessed via multiple different paths (e.g., the same object is read via `*p` and written via `*q`, both inside the same iteration of the same MU). 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 reads of 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 essential to ensure no writes to exterior objects be 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 a way that it computes 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 then store in a Temporary Cache (TC) instead of writing to the exterior object. Of course this means, 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. To generate the IV, we simply store all reads on exterior 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

```
#include <memoize>
void fib[[]](::std::uint64_t *in) {
  // the anchor is already hardcoded and need not be memoized
  if (*in <= 1) return;
  /* transformed code begins here */ {
#ifndef _STD_MEMOIZE_ENABLE
#define _STD_MEMOIZE_ENABLE true
#endif
    if(_STD_MEMOIZE_ENABLE) {
      try {
        auto __policy = ::std::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::memoize::at_scope_exit([&] {
            __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::memoize::alias_exception const&) {
        /** -- original code here (Fig. 2, lines 6-10) -- **/
      }
    } else { /** -- original code here (Fig. 2, lines 6-10) -- **/ }
  } /* transformed code ends here */
}
```

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

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.

After adding the interceptions, we eliminate all code that does not eventually contribute to the IV. Since the possibility of aliasing would prevent a lot of code from being eliminated, our dead code elimination assumes no aliasing occurred and we ensure this property at runtime by way of the TC. The TC provides an overlay address space that not only stores write operations during IV computation but also recognizes whether two paths point to identical or overlapping memory, i.e., alias. If aliasing occurs, e.g., since the same object is referenced multiple times via different ways through a cyclic data structure, then we abort the memoization. Details on the dead code elimination and the TC are provided in the online appendix and Reference [28].

3.4 Performing Memoization

Next, we need to determine whether the same input has occurred before. This 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 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 no aliases are encountered, then 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.

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, among 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 necessarily performs all writes. Hence, 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 is already 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 Memoization Decision

As discussed in Section 2.4 we provide developers with an opportunity to decide whether memoization should be applied in a certain circumstance or not. Consider the Fibonacci example shown in Figure 2. As the original function is computationally extremely simple, the overhead of memoization only pays off if the requested Fibonacci number is big enough. The actual break-even-point can be easily determined by performance measurements and has been determined to be about 18 for our configuration. For smaller values, the original code provides better performance.

To support this, the developer can provide a macro that is evaluated at the beginning of the transformed MU. The default implementation of this macro always issues `true`, hence always opting for memoization. For the described use case, it makes sense to define this macro as the expression `*in > 18`. On execution of the transformed MU, we evaluate this macro and perform memoization exactly if it yields `true`.

By providing a way to tune the memoization behavior further, we hope to reduce the performance gap between an automated general purpose transformation and a careful manual implementation. Our approach works fine without this manual effort, but performance can be improved by proper usage of this feature.

3.7 Cache Eviction

Without evicting entries the MC grows with every new input. While this is fine in cases where the memory usage is unproblematic—which is often the case on modern computers—we still need a solution that copes with memory limitations. To demonstrate the versatility of our approach, we implemented the well-known cache eviction strategies Random Replacement (RR) and Least Recently Used (LRU). Either of them can be easily activated and configured for different MC sizes. Additional strategies can also be implemented by the model developer.

The RR strategy randomly selects an entry to remove on insertion of a new entry into the MC if the size exceeds the specified limit. This strategy works at least as well as the more complex LRU scheme if the number of entries is significantly greater than the number of recently used entries, as it becomes improbable to evict an entry too quickly. The LRU strategy selects the entry that was least recently used on occurrence of a new entry to be evicted from the MC. Usually the RR strategy is implemented in a way that has a smaller overhead but leads to poorer results. Due to the difficulty of choosing a random element from an unordered map, the overhead of random replacement is comparatively high in our case, while we managed to implement the LRU strategy in an efficient manner by linking the entries in a doubly-linked list.

When a new entry is added to the MC, it is, by definition, the most recently used element, hence it is always enqueued at the beginning of the list in $O(1)$. Removing the least recently used element is accomplished in constant time as well, since the element is found at the end of the list. When an existing entry is used, this entry can be anywhere in the list, but is found in constant time, since the list node is tied to the MC item as explained above. This entry then needs to be shifted to the beginning of the list. To this end it is removed from the list structure at its current location ($O(1)$) and enqueued at the beginning ($O(1)$). Hence, the LRU strategy adds a constant time overhead that is independent of the lookup complexity for entries in the MC, beyond the trivially necessary singular insertion and—where necessary—deletion.

Either technique is implemented as a class template providing a thin wrapper around an underlying dictionary. The underlying dictionary is provided as a type parameter and only required to provide a subset of the `::std::unordered_map` API. As our research prototype allows the developer to specify the dictionary type used for the MC themselves, usage of cache eviction only requires providing the appropriate evicting dictionary. Additional strategies—including any that rely on domain knowledge—may be implemented using the same faculties and require neither the memoization transformation to be changed, nor modification of our library component.

3.8 Dictionary Serialization

The developer can provide the path to a file that the MC is written to at the end of the execution as well as read from at the beginning (if it exists). This way we support a simple way to implement parameter studies where each configuration is executed in a separate process. By using the same

file for all iterations, the MC can thus be used to memoize across different configurations. To this end, we implemented serialization and deserialization for our data structures.

4 EVALUATION

We evaluate our approach by first performing measurements to characterize the overhead of our approach. We measure the overhead of the memoization itself (separately for cache hits and cache misses) as well as the overhead of storing and retrieving the MC to and from disk. Second, we implement the well-known Fibonacci-computation in a recursive implementation gratuitously using pointers as an illustrative example to show the feasibility of our approach on recursive and pointer-based code and validate the performance characteristics. In addition, we use this benchmark to analyze the impact of coexistence and cache eviction. Third, we investigate the applicability of automated memoization to random number generation, a feature used in almost every simulation. Finally, we analyze the practical applicability of our approach in modeling and simulation by way of three case-studies from different domains: (1) a case study performing a parameter study of an OFDM (orthogonal frequency-division multiplexing) network simulated by OMNeT++ [31], (2) the 802.11p decider of the Vehicular Ad-Hoc Network (VANET) simulator Veins, and (3) a predator-prey-model following the Arditi-Ginzburg equations. In all examples aliasing does not occur inside the MU.

All measurements are performed on a Xeon E5 compute server with 256GB of RAM with no swap space enabled. Each experiment is repeated 10 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. Besides the new experiments performed for this article, we repeated the ones we performed in Reference [29] on the new evaluation platform and with all software improvements included. This ensures that all results in this article are directly comparable.

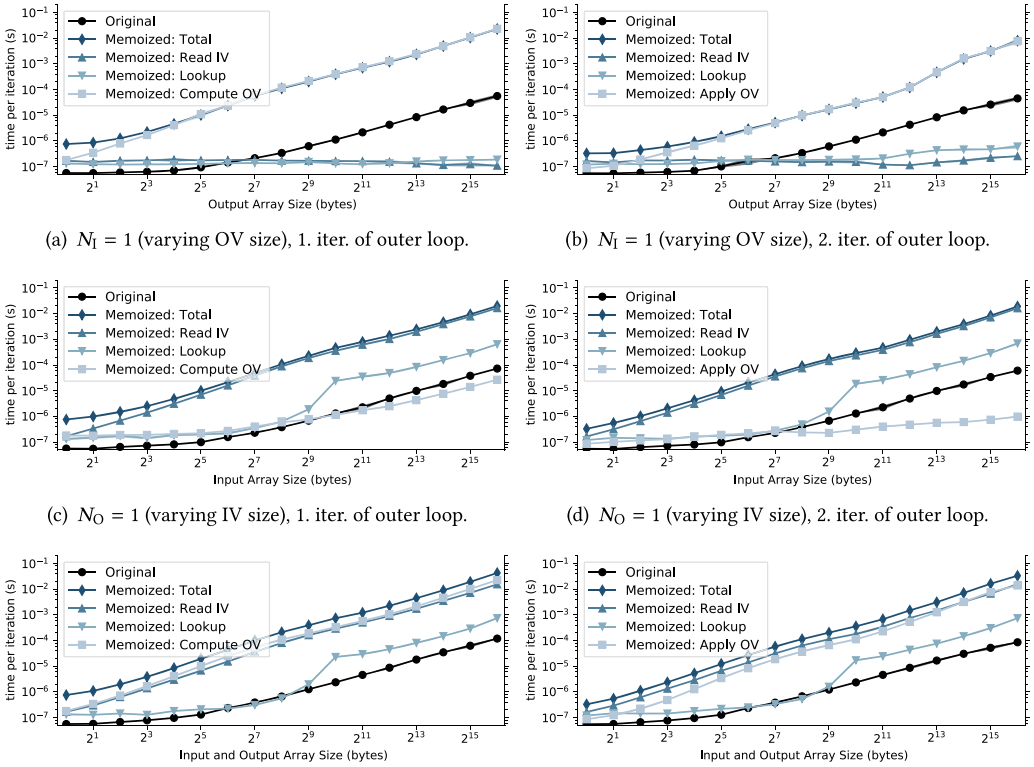
As any optimization only makes sense if the results are not just computed quickly but also correctly, we validated the output of every model. In each experiment we compared the computational results of the memoized version against the results of the original implementation. The results were identical in every case. Hence, we conclude that for every experiment performed in this article the transformation was performed correctly and maintained the semantics of the original program.

4.1 Overhead Evaluation

First, it is necessary to understand how much overhead automated memoization introduces to be able to investigate how much gain must be yielded to benefit. Our approach induces overhead during execution of the MU itself as well as during initialization and teardown when the MC should be retrieved from and saved to disk.

We measure both types of overhead by means of a simple synthetic benchmark: An array of N_I 8-bit numbers is read, the numbers are aggregated into a 64-bit variable, which is then xor-folded to yield an 8-bit result. An array of N_O 8-bit integers 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.



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

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

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

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

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

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

Fig. 4. Synthetic benchmark where memoization cannot gain benefit. Used to measure the overhead.

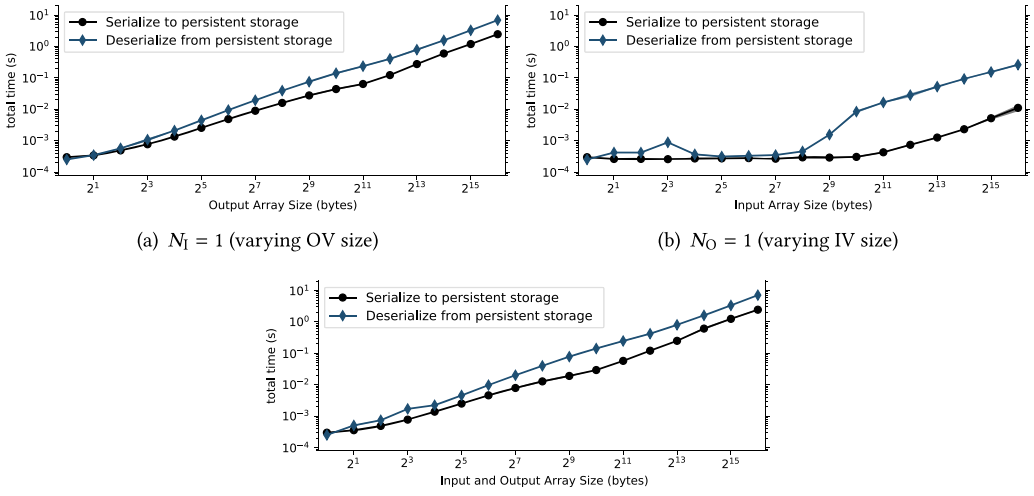
We performed experiments while growing only N_I, only N_O, and growing both simultaneously. To evaluate the runtime overhead we disabled persistence and enabled it thereafter to measure the serialization and deserialization overhead.

Runtime Overhead. Figure 4 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 contributes very little to the total runtime (note the logarithmic scale). In the first iteration, for large output arrays the overall runtime is almost equivalent to the runtime of the OV computation whose cost primarily depends on N_O. For smaller output arrays additionally the IV reading becomes relevant.

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. For the application of the OV the overhead is linear in N_O but much smaller than for the computation of the OV. Furthermore, we observed (without figure) memory overhead linear in both N_I and N_O.

Serialization and Deserialization Overhead. To measure the overhead induced by storing the MC to disk and loading it in a later program execution, we enable persistence and execute the benchmark twice. Hence, in the first execution the MC is stored to disk and loaded in the second execution. We note that the loading time highly depends on external circumstances beyond our control,



(a) $N_I = 1$ (varying OV size)

(b) $N_O = 1$ (varying IV size)

(c) $N_I = N_O$ (IV, OV varying)

Fig. 5. Overhead evaluation: serialization and deserialization of Memoization Cache.

in particular caused by file system caches maintained by the OS, hardware cache included in the drive as well as storage performance itself. In this context, our benchmarks aim to mimic a common use case: In an exploratory parameter study, different experiments are conducted, results are analyzed, and more experiments are launched. To this end, we perform the first execution of all configurations first (creating MC in the file system), then execute the unmemoized benchmark (causing system load that does not stem from the MC files), and, finally, execute the memoized configurations again (retrieving the files from disk). It is likely that the MC files are already present in the file system cache in our benchmark, such that actual performance might be lower if the files have to be loaded from disk. However, for the common case of exploratory parameter studies described above we expect the situation to be similar to our benchmark and we argue that the actual performance of our drive is of limited general interest.

We measure the time to persist the MC to disk from the point we start serialization up to the point the OS signals completion of the file close operation. This might not coincide with the point the data is actually stored physically in persistent storage but coincides with the point where users perceive completion and can continue their work.

Figure 5 depicts the measured serialization and deserialization times for persisting and retrieving the MC with all 250 entries, with a varying number of entries. For the smallest IV and OV size of 1B each, we observe serialization and deserialization time to be both about 0.3ms. With increasing OV size, we observe linear runtime increase for both serialization and deserialization. The influence of the IV size is much lesser. Up to 256B no significant increase can be observed, from 512B on we find a linear increase, but a byte of input has orders of magnitude less influence on the runtime than a byte of output.

This behavior can be easily explained: For the IV, the MC only stores the actual data values, the path via which the values are to be retrieved is given by the code. For the OV, however, the result application routine needs to know *where* to store the values found in the OV. Hence, for every value (in this benchmark each value has a size of 1B) we need to store the value *and* the path. This creates more data to store on disk and more complex data structures to be both serialized and deserialized.

This observation is confirmed by investigating size and structure of the MC file (without figure): The MC file contains an 8B header giving the number of entries in the cache. For each entry, the file contains 8B header for the IV and 8B for the OV to specify the number of entries in the respective vector. For an MC with 250 entries this yields 4,008B. Additionally, the IV itself cons umes 1B for each byte of input. For the OV, however, our implementation requires 13B for each byte of output, i.e., the output byte plus 12B to store the path. Hence, for 1B of both input and output the file size is 7,508B. For 2^{16}B of input and 1B of output the file consumes about 16MB of disk space while it consumes more than 200MB for 1B of input and 2^{16}B of output.

It should be noted that this format, while reasonable, is by no means optimal, and could be improved in various ways, e.g., by compressing path representations or storing a checksum of the MU for which the file is valid.

4.2 Recursive Fibonacci with Pointers

To demonstrate the potential benefit of memoization and the versatility of our approach, we implement a common showcase for memoization. Note that though this example is not practically relevant in the modeling and simulation domain, it is quite intuitive and facilitates understanding the impact of different conditions on the performance of our approach. We implement a naïve recursive function (see Figure 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 Section 2.2) and annotated accordingly to memoize this recursive function. It must be noted that, should the IV generation not be optimized well enough (cf. Section 3.3), this function would perform the whole computation before any memoization takes place.

In Reference [29] we observe the expected exponential runtime of the unmemoized algorithm and the linear runtime of the memoized version until the memory limit is hit. However, we also observe that the memoization overhead is bigger than the gain for very small Fibonacci numbers. The techniques discussed in Section 3.6 (coexistence of and runtime-switching between original and memoized code) as well as Section 3.7 (cache eviction) seem promising for this purpose. We discuss their impact on this benchmark in the following.

Coexistence and Runtime Switching. The results discussed so far reveal that memoization is only beneficial for Fibonacci numbers from approximately 18 on. As discussed in Section 3.6 we provide the opportunity to decide at runtime whether to apply memoization or not, based on arbitrary, user-defined conditions. Defining such a condition is as easy as defining a macro that will afterwards be evaluated in a truth context. For our benchmark, the macro we need to provide is simply `*in > 18`, i.e., memoization shall be applied exactly if the requested Fibonacci number is greater than 18.

Figure 6 depicts the results. The general shape of the performance curve is as predicted: Up to F_{18}, it follows the performance of the original curve, and at F_{18} it changes its slope and follows the memoized performance curve. Hence, for any input it follows the lower curve and provides about the performance of the better of the two approaches. However, we observe as well that memoization with coexistence does not entirely reach the performance of the version without. We attribute this to two facts, caused by the addition of a conditional and a code block that is—for the given input—never executed: First, the conditional itself has to be evaluated, which costs about 3 cycles on the target machine. Second, the added code poses challenges to the optimizer of the compiler that is invoked after our source-to-source translation and causes it to generate a less efficient

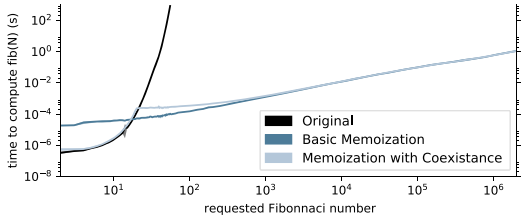
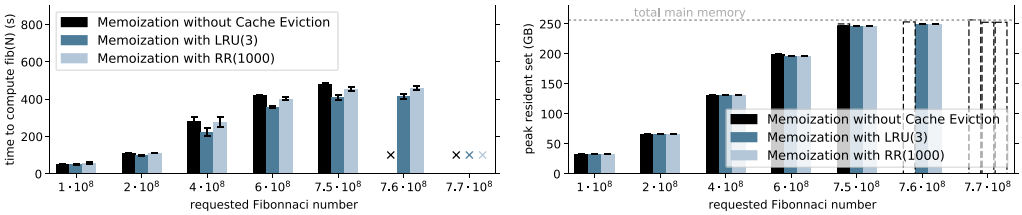


Fig. 6. Performance improvements achieved for recursive Fibonacci computation by activating coexistence.



(a) Runtime. Crosses: Benchmark did not finish due to insufficient memory.

(b) Memory consumption. Dotted line: physical main memory. Dashed bars: linear projection on potential memory consumption

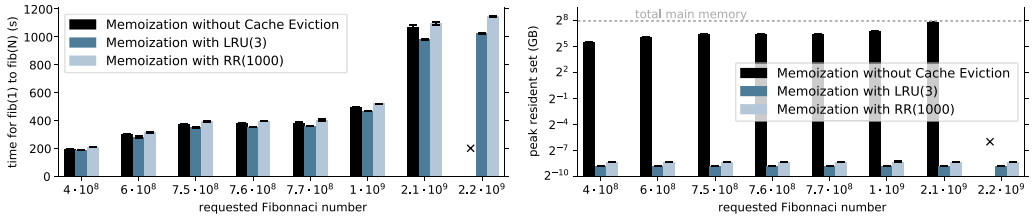
Fig. 7. Performance of recursive Fibonacci computation with cache eviction.

register schedule. This roughly costs an additional 19 cycles on the target machine. Note that these costs are cumulative in the sense that the conditional needs to be evaluated each time the function is (recursively) called. The effect of these problems is quite significant for numbers between 18 and about 100 while the performance approaches the performance of basic memoization when the requested Fibonacci number grows. We conclude that using the opportunity of runtime switching is promising if each of the two versions is particularly beneficial in a significant number of cases.

Cache Eviction. Cache eviction can help limit the MC's memory consumption. In our benchmark, the nth Fibonacci number is computed by adding the $(n-1)$th and the $(n-2)$th Fibonacci number. Hence, it is actually unnecessary to hold all previously computed Fibonacci numbers in the cache, but at any point in time we only need three of them: F_n, F_{n-1}, and F_{n-2}. These are also the three most recently used numbers in this experiment, so we identified LRU(3) as a promising cache eviction strategy. Additionally, we investigate RR(1000), which has a low probability to evict one of the required entries (hence forcing re-computation of that entry). Though the MC is somewhat larger for RR (1000 entries instead of 3), memory requirements are still fairly low.

The results are depicted in Figure 7(a). Up to $F_{7.5\cdot10^8}$, we observe the linear behavior we encountered before. We also observe some performance improvements by cache eviction, since lookups perform faster if the MC is smaller. We attribute this to the fact that even though in theory hash maps perform all operations we use in constant time complexity, in practice growing the MC does have a (small) negative performance impact. Additionally, LRU performs faster than RR, which we attribute to the fact that though RR(1000) in most cases evicts entries that are no longer required, with every eviction operation there is a small probability to evict an entry that was required and must be re-computed later. We also observe that the bookkeeping overhead of LRU(3) does not pose a performance bottleneck in this particular benchmark.

For $F_{7.6\cdot10^8}$, without cache eviction, the program no longer fits into memory and is terminated, while LRU and RR keep the memory consumption in the bounds of available main memory. Figure 7(b) shows the total memory consumed: The peak resident set grows linearly until it reaches the



(a) Runtime. (b) Memory consumption.

Fig. 8. Performance impact of cache eviction on a benchmark that recursively computes a sequence of Fibonacci numbers. Crosses: Did not finish due to insufficient memory.

physical boundary of available memory. The linear projection shows that for $F_{7.5\cdot10^8}$ the resident set would grow larger if more memory was available, but the usual memory-pressure measures ensure the program completes within the bounds of available memory. For $F_{7.6\cdot10^8}$, the system can no longer satisfy the memory requirements. However, contrary to the first expectations, for $F_{7.7\cdot10^8}$ we also observe the program being killed with either of the cache eviction strategies applied. We found the reason for this behavior in the way the experiment is designed:

Starting with the requested number, the benchmark recursively computes all lower Fibonacci numbers. While the original implementation recurses into two directions (F_{n-1} and F_{n-2}) and builds up a recursion tree of exponential size, memoization can avoid stepping down in both directions, but still requires to step down in one direction. Hence, it results in a recursion path of length n, when F_n is requested. At the *beginning* of each function, a TC has to be created (as it stores all reads during execution, see Section 3.3), while the MC entry is only created at the *end* of each function (obviously, this can only be done *after* the result has been computed). Hence, while computing F_n, about n TCs have to be kept in memory in addition to the already sizable stack frames, until F_1 has been reached. Then, the result of F_1 is stored in the MC, followed by F_2, F_3, and so on. We conclude that the memory requirements of this benchmark are in majority *not* caused by the size of the MC, hence cache eviction, for this particular benchmark, is only of limited use.

Modified Benchmark: Enumerating Fibonacci Numbers. To analyze the full potential of cache eviction, we modify the benchmark slightly: In the nth experiment, instead of computing only F_n, we compute $\sum_{i=1}^{n} F_i$ (again in the residue class of the modulo 2^{64}), such that all Fibonacci numbers from F_1 up to F_n are computed sequentially. By not computing F_n on an empty MC, but instead after having computed F_{n-1} and F_{n-2} before, we avoid the necessity to execute the entire recursion path down to F_1 as the MC already holds F_{n-1} and F_{n-2}. Thus we expect constant time complexity for computing a single Fibonacci number when the preceding numbers have been computed before, but—as we are computing n Fibonacci numbers this time—overall performance is again linear in n. The number of TCs created during the execution is linear in n as well, but at any point in time only a single TC exists in memory, significantly reducing peak memory consumption. Hence, this benchmark captures the possible impact of cache eviction much more clearly. The unmemoized implementation does not benefit from the fact that preceding Fibonacci numbers have been computed before and performs even worse in this benchmark. For this reason, we omit a discussion of the unmemoized implementation.

From the results depicted in Figure 8 we observe that memory consumption is significantly decreased with and without cache eviction: In fact, even without cache eviction we find a maximum resident set of only 93GB for computing Fibonacci numbers up to $F_{7.7\cdot10^8}$. The MC then hits the memory limits at $F_{2.2\cdot10^9}$. At the same time, we observe only about 2-3MB of memory being used by the entire process when either cache eviction strategy is active. The total memory consumption

is slightly higher for RR(1000), since more entries are stored, but the amount of memory hold is not significant in either case.

We conclude that memoization can significantly reduce the runtime of this benchmark. Coexistence provides improvements in cases where memoization can not yield a benefit. The size of the MC can be effectively kept at a very low level by means of cache eviction strategies. This prevents excessive memory consumption, at least, if the memory consumption is in fact caused by the MC and not by a huge number of TCs due to deep recursion.

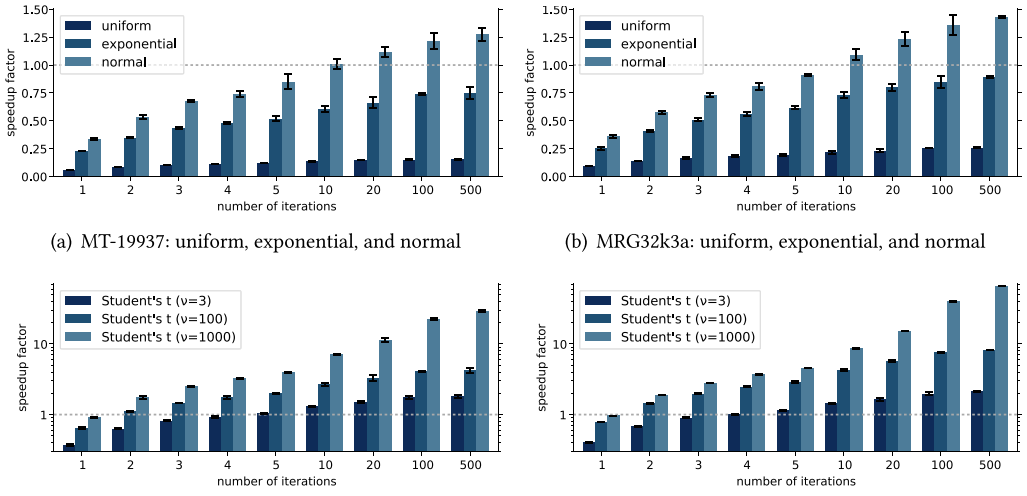
4.3 Random Number Generation

An essential component of virtually every simulation is random number generation. To enable randomness, but at the same time maintain repeatability, Pseudo Random Number Generators (PRNGs) are typically used. After being seeded with a user-provided value, those PRNGs issue a random-looking yet deterministic sequence of numbers. Internally, this works by generating a state during seeding that only depends on the seed, and transforming this state deterministically into a new state by a predefined function on every request of a random number. This random number then depends only on the current PRNG state and follows a predefined distribution, which is typically a uniform distribution over all values a certain type (e.g., `uint32_t` or `uint64_t`) can hold. To still be able to provide other distributions (e.g., exponential distribution with a given mean, or normal distribution with a given mean and standard deviation), one or more numbers drawn from the PRNG are converted by another deterministic function into a number following the desired distribution.

As random number generation is required in any stochastic simulation, it appears to be a promising target for memoization. A simulation could be sped up if the transformation of the current state into a new one plus a random number and the conversion of the random number into a random variable from a given distribution could be avoided. Unfortunately, the rather large states many PRNGs maintain (e.g., 2.5KB for Mersenne Twister 19937 [16]) and the fact that the complexity of the state transformation is, compared to the size of the state, rather low, reduce the potential benefit. By transforming the problem slightly, memoization can yield more benefit without a necessity to modify the PRNG interface towards the simulation model: Instead of implementing the PRNG as a function that maps a current state to a next state and a random number, we implement it as a function that internally maintains an *ID* of the number generated lastly and increases this ID by one on every execution. It then calls the PRNG to advance the state associated with the requested stream to the state corresponding to the given ID. Without memoization applied this always advances the PRNG from the state preceding the ID to the one corresponding to the ID, i.e., it is equivalent to what PRNGs usually do. With memoization applied, however, only the ID needs to be increased and the result can be returned immediately. The memoization then identifies this ID and the stream identifier as input and emits the increased ID and the returned number as output. Hence, when two experiments with potentially different parameters but the same PRNG seed are run, random numbers can be retrieved from the MC.

Nevertheless, the complexity of the operations performed inside a PRNG is rather low, such that the overhead of memoization would not pay off. By additionally including the distribution transformation operation, the complexity is slightly increased, and memoization can pay off in certain cases. However, speedup can only be expected if the transformation itself bears considerable complexity.

We evaluate the performance with four different types of distributions and two different PRNGs. We chose the PRNGs Mersenne Twister 19937 [16] and MRG32k3a [15] used by OMNeT++ and ns-3, respectively. We use the distribution transformation code from OMNeT++ (`distrib.cc`) and selected uniform (equivalent to the output of the PRNGs), exponential (essentially computing a



(a) MT-19937: uniform, exponential, and normal

(b) MRG32k3a: uniform, exponential, and normal

(c) MT-19937: Student's t distribution

(d) MRG32k3a: Student's t distribution

Fig. 9. Performance of memoizing random number generation with different PRNGs and distributions.

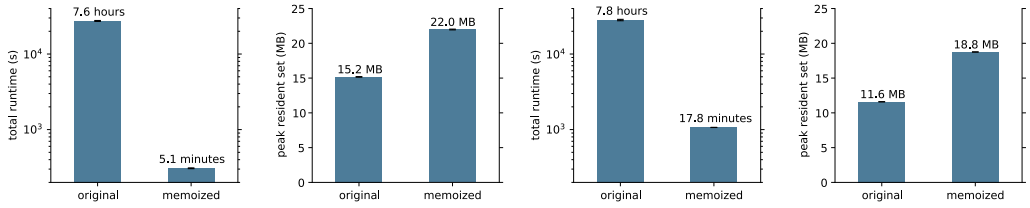
logarithm), normal (using two numbers and applying logarithm, cosine, and square root), and Student's t distribution. The complexity of the former three is quite low and independent of the parameterization. The complexity of the latter increases with the degrees of freedom (ν). Hence, we selected three different parameters for Student's t distribution. We varied the number of iterations from 1 (no reuse possible) up to 500 (each number can be reused 499 times by memoization).

The results of our experiments are depicted in Figure 9. As expected, for the computationally simple distributions uniform and exponential memoization cannot gain a benefit, and should not be used to avoid slowdowns. However, for a normal distribution, which just draws two random numbers and performs a logarithm, a cosine, and a square root on them, automated memoization can already gain a speedup after reusing the results 10 times. The creation of MC entries causing the computation to slow down by a factor of 3 in the first iteration, pays off if just 10 experiments are run with the same PRNG seed. For Student's t distribution with 3 degrees of freedom memoization already pays off in the fifth iteration. With more degrees of freedom, Student's t distribution becomes a particularly computationally expensive problem and the memoization overhead in the first iteration becomes almost negligible. Memoization then already pays off from the second iteration on.

We conclude that although the low complexity of random number generation is challenging for memoization, our approach can still yield a benefit in certain cases. However, for uniformly or exponentially distributed random numbers, memoization can only pay off if the random number generation is part of a larger MU with more computational complexity, or if less efficient random number generators are used (e.g., when using generators that provide additional security guarantees).

4.4 OFDM Wireless 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++ [31] in C++. In the simulation model, a set of wireless nodes is placed on a 1 by 1km area. A number of those nodes transmits frames in a fixed pattern. A channel model based on the Friis path loss model [6] and a complex OFDM fading model [33] calculates the received signal strength. In the parameter study, the total number of wireless nodes is varied



(a) Single-Process Study. Runtime. (b) Single-Process Study. Memory. (c) Multiple Processes. Runtime. (d) Multiple Processes. Memory.

Fig. 10. Performance of OFDM Model run in a single process or multiple processes with MC persistence.

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 1µs to 10ms. Each experiment is repeated 10 times with different random number generator seeds, hence the total parameter study consists of 6,300 runs.

We identified the fading computation as a promising candidate for memoization as it is a complex operation and its input is small and repeated frequently. This fading computation heavily uses pointers to iterate over multi-dimensional arrays. Prior approaches to automated memoization would treat these pointers by their address and 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++ 5.1.1 on the above-described hardware.

OMNeT++ allows us to specify the entire parameter study via an initialization file. If the user does so, then OMNeT++ parses the file during program startup and executes all configurations sequentially in a single process. Between the execution of two configurations, OMNeT++ tears down the simulation model but does not quit the OS process. Hence, memory that is not explicitly cleared is still available in the next configuration, and memoization can yield benefit across runs as an entry stored in the MC is still available to the next configuration.

However, specifying the entire parameter study at the beginning might not always be applicable for several reasons: The language might be too restricted to specify the configurations of interest adequately, or the user might be simply not confident enough to do so and hence prefer launching the configurations individually by a script. Furthermore, this execution mode requires clean teardown methods and is less tolerant to memory leaks as leakage sums up over the runtime of the entire study. Our concept of automated memoization is not only designed for OMNeT++, but as well for simulation frameworks that simply do not provide such an operation mode. Finally, in exploratory parameter studies users decide which configurations to run next based on the results of the previous execution, and hence it is simply not possible for them to specify the entire study in the beginning.

If the simulation study cannot be run as a single process for any of the aforementioned reasons, then automated memoization can still gain benefit if the MC is stored to disk after each run and loaded at the beginning of the next run. Obviously, this introduces additional overhead but might still gain benefit. We therefore decided to specify the study (1) by means of the OMNeT++ initialization file to be run as a single process and (2) via a shell script that launches multiple processes sequentially. We measured the total runtime of both operation modes to compare the speedups gained.

Single-Process Parameter Study. The results of the first mode where the parameter study is executed as a single process are depicted in Figure 10(a) and (b). In the original implementation, each

run took about 4-5s, resulting in a total runtime of about 8 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 onward, we observed significant speedups, certain runs were completed in as little as 5ms. Completing the total parameter study then takes about 5 minutes, hence our automated memoization yielded a speedup of more than 80×.

The MC, on the other hand, increased the memory consumption of the program by 40% from 15MB to 22MB. We feel confident asserting that a penalty of about 7MB of memory will be happily accepted by a user who now only has to wait minutes instead of hours.

Multi-Process Parameter Study. When executing the parameter study by launching each configuration as its own process via a shell script (results depicted in Figure 10(c) and (d)), the runtime of the memoized parameter study is tripled, compared to the single-process version. There are two sources for the increase of runtime: First, the MC has to be loaded from and stored to disk at the beginning and end of each configuration. Second, OMNeT++ itself has to be started and stopped at the beginning and end of each configuration. While the first effect is only present in the memoized execution, the second effect affects the original execution as well. However, the effort of starting and stopping OMNeT++ is constant while the actual runtime of an experiment is reduced from 4-5s to 5ms, increasing the execution time of both studies by more than 10 minutes.

To determine the peak resident set of the parameter study we measured the memory footprint of each process and selected the one consuming the most memory. The total memory footprint of both original and memoized execution is reduced by 3MB, potentially due to memory (leakage) that sums up during the execution of the parameter study, but is in the latter case collected by the OS when a process ends. However, memoization still adds the same memory increase of 7MB. Additionally, the persisted MC consumes about 2MB of disk space.

We conclude that memoization gains significant speedup for this parameter study in both single-process and multi-process mode. In single-process mode a much higher gain is achieved as the overhead of both starting/stopping OMNeT++ and deserializing/serializing the MC is quite significant in multi-process mode. Nevertheless, automated memoization still proves highly valuable for exploratory parameter studies or other use cases where a multi-process mode has to be used. In both modes, the memory footprint is increased by just 7MB and the persisted MC takes 2MB of disk space, which we deem a very acceptable price for a runtime reduction of several hours.

4.5 Vehicular Network Simulator Veins

Vehicles in Network Simulation (Veins) [27] is a car2x communication simulation framework for OMNeT++. It is available as on open source project, actively developed by several universities and research labs and frequently used by all kinds of institutions from academia to industry from all over the world [26]. We analyzed the source-code and identified the function `getChunkSuccessRate` as a very core component: It is invoked every time a packet is received to decide if the packet PHY header and the packet payload are successfully transmitted. The function returns a success rate, which deterministically depends on data-rate, bandwidth, packet/header length, and signal-to-interference-and-noise ratio (SINR). For the PHY header the former three are always identical and only the SINR deviates. Even for payloads a high reuse can be expected, since in a typical VANET scenario many packets have the same length and only few different data-rates and bandwidths are used. However, since Veins is used by many projects in different ways we focused our measurements on the settings of the PHY header, which are identical in any use case.

We base our experiments on Veins 4.6. We wrapped the computation of the success rate depending on the modulation and coding scheme (MCS) into a compound statement and annotated this

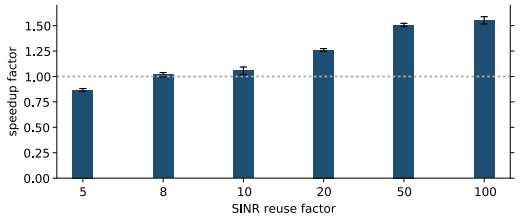


Fig. 11. Speedup achieved by memoizing the computation of the header success rate in Veins. Memoization pays off if at least 8 packets are sent under the same SINR conditions.

block as the MU. To keep our experiments independent of concrete use cases of Veins, we created a benchmark that calls the function `getChunkSuccessRate` repeatedly. The data-rate, bandwidth, and length are set as in the PHY header (3MBit/s, 10MHz, 24Bit), for the SINR we generated uniformly distributed floating point numbers between 0 and 3 to include ranges in which the success rate is 0, almost 1, and in between. We call the function 5 million times and vary the number of different SNRs from 50,000 to 1 million, i.e., the same SNR is used 5 to 100 times. We maximize the distance between two calls with same input to minimize positive caching effects. Thereby, we determine how often the function need be called with the same input for memoization to pay off.

Figure 11 shows the speedup factor achieved in the different configurations. We observe that already for a reuse factor of 8, i.e., when 8 packets are sent under the same SINR conditions, the memoization gain is slightly higher than the overhead. When the reuse increases, the speedup increases up to a factor of 1.5. While we measured this for the parameters used for computing the header success rate, similar results can be expected for the payload success rate if SINR, data-rate, bandwidth, and packet length are identical. The speedup might be even a bit higher, since the computations are more complex for FEC 3/4 than for FEC 1/2. Compared to the first network simulation case study the speedup here is quite low, which is caused by the relatively low complexity of the success rate computation. More detailed channel models with higher computational complexity obviously maintain higher optimization potential. However, we observe that even such a simple decision component can be sped up by memoization.

4.6 Predator-Prey Model

Predator-prey models are used to simulate the fluctuation in size of two populations where one species feeds on the other. The first species, the prey, grows as described in a predetermined function (birth rate) if no individuals of the second species, the predators, are present. If predators are present, then individuals of the first species are killed as determined by another function (feed rate), hence the population shrinks if the feed rate is greater than the birth rate. Finally, the growth rate of the predator species depends on the current size of the prey species, since predators can only survive if enough food is available.

We created a model following the Arditi-Ginzburg equations [3]. These are a pair of differential equations that describe the change of the two populations. The size change of the prey population depends on the size of both populations and two functions to be determined: the "per capita rate of increase of the prey in the absence of predation" $f(N)$, and the "predator-dependent trophic function" $g(N, P)$ with the current prey population N and current predator population P. The size change of the predator population depends on the size of both populations, a function, and a constant: the "predator production per capita" $h(N, P)$ and the constant "food-independent predator mortality" μ.

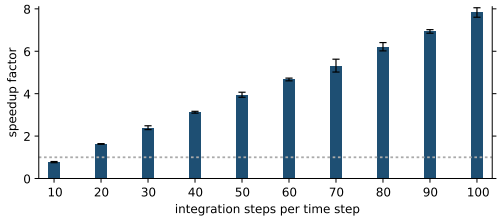


Fig. 12. Speedup gained by memoization in an Arditi-Ginzburg-based predator-prey model. The speedup increases linearly with increasing number of integration steps.

We define the functions and the constant as follows: $f(N) = 0.3 - 3 \cdot 10^{-6} N$, $g(N, P) = {}^{0.82N}/_{(N+P)}$, $h(N, P) = 0.78 g(N, P)$, $\mu = 0.5$. We implement a time-step simulation, i.e., the differential equations are numerically integrated in each time step. The numerical integration has a variable step-size and the result accuracy increases with the number of integration steps per time step. Likewise, the computational complexity increases and hence the potential memoization gain. We memoize the integration per time unit, i.e., the input are pointers to the populations at the beginning of each time step, which are updated during the computation.

In our experiments we ran the simulation for 10 million units of time and varied the number of integration steps per time unit from 10 to 100. The memoization speedup is depicted in Figure 12. For 10 integration steps the computational complexity is too low to repay the memoization overhead. However, the memoization overhead stays constant and the speedup factor increases linear with an increasing number of integration steps. For 20 integration steps the speedup is already almost twofold, for 100 steps it is almost eightfold. We expect similar performance for other simulations based on numerical integration and even higher gains if the underlying differential equation system is more complex than the rather simple one used here. We conclude that memoization provides speedups in simulations of very different domains.

5 RELATED WORK

Memoization was first introduced by Michie [19, 20] and implemented in a framework by Popplestone [23] 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 [21] provide an in-depth analysis of the memoization idea and the resulting challenges such as 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 to automated memoization have been proposed, e.g., the ones by Norvig [22], Hall et al. [7, 17, 18], and Hinze [9]. They base on Haskell and Lisp, but also C++. 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 [18], which effectively adopts the Lisp approach from Reference [17]. Hence, input may only be provided in function parameters, only the return value may be output, and pointers are 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 that object is actually input. In logic programming languages the concept of tabling is used to memoize results of previously evaluated (by definition pure) rules [35]. To the best of our knowledge,

no generic approach to automated memoization without restriction to pure functions has been proposed previously.

A central element of memoization for impure code is the identification of input and output, i.e., relevant data items have to be identified in the memory map and distinguished from irrelevant ones. A similar problem exists in transparent incremental state saving [4, 24, 34] as well as automatic generation of reversible code [25]. While it seems promising on the first glance to adopt those techniques to identify input and output of the MU, there is a major difference in the assumptions that can be made with respect to pointers: An object found at address *a* during state saving or forward computation can and needs to be restored at address *a* during rollback, meaning it is identified by its memory location. For automated memoization, however, memory location is neither sufficient nor required to ensure equality. Furthermore, self-adjusting computation [1, 2] faces similar problems. However, these approaches do not operate on existing code (since tracking the dependencies is too expensive for their purposes) but rather require the user to use so-called *modifiable references*, whose reading and writing is to be explicitly notified by additional code. For this reason, we developed a new approach to identify input and output and decided to take the approach to describe an object by its path through the object graph as discussed in Section 2.2.

Our source-to-source transformation realizes the opportunity to switch between memoized and non-memoized versions of the same code at runtime via a straightforward conditional provided by the developer. Dual-coding [32] should be mentioned as an alternative that adds this feature during binary translation and aims at reducing the overhead by minimizing the number of branches. This is particularly beneficial in the use case dual-coding has been developed for where a branch would be induced on every memory access. However, in our use case we branch only once at the entry of the MU, which also provides the user with maximum flexibility when deciding whether to use memoization or not on each execution of the MU.

Tsumura et al. [13, 30] propose to integrate memoization functionality directly into the processor hardware. While this is probably the most promising approach to speed up any software independently 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 be practically applicable in the modeling and simulation domain.

To avoid unnecessary computations in simulation parameter studies, simulation cloning [10, 11] and updateable simulations [5] 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, cannot 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

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 being re-computed.

In this article, 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++, and first

discussed in Reference [29]. Our approach provides an automated transformation of a user annotated code block into an equivalent, memoized version. Additionally, the original code is included and the user is provided an opportunity to select at runtime which version to use. To keep the size of the MC maintainable, the number of entries can be limited and common cache eviction strategies applied. Furthermore, the MC can be serialized to disk and reloaded in a later execution to enable reusing computation results across multiple executions.

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. Detailed overhead measurements are provided in Section 4.1. In a simulation parameter study we observed a more than 80× speedup while only increasing memory consumption by about 7MB. Additionally, we demonstrate the feasibility to implement extensions like cache eviction to tune the memoization for a given situation. 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. Furthermore, our approach needs to cope with multithreaded software, enabling multiple threads to concurrently and cooperatively utilize a common MC, which is by now only possible to a limited extent. This will combine the power of both PDES and memoization to benefit from both. Finally, the performance of the proof-of-concept implementation can still 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 even with computations of rather low complexity.

REFERENCES

[1] Umut A. Acar. 2009. Self-adjusting computation: (An overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 1–6.

[2] Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative self-adjusting computation. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 309–322.

[3] Roger Arditi and Lev R. Ginzburg. 1989. Coupling in predator-prey dynamics: Ratio-dependence. *J. Theoret. Biol.* 139, 3 (1989), 311–326.

[4] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2015. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 211–222.

[5] Steve Ferenci, Richard Fujimoto, Mostafa Ammar, Kalyan Perumalla, and George Riley. 2002. Updateable simulation of communication networks. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*. 107–114.

[6] Harald Friis. 1946. A note on a simple transmission formula. *Proc. Inst. Radio Eng.* 34, 5 (1946), 254–256.

[7] Marty Hall and James Mayfield. 1993. Improving the performance of AI software: Payoffs and pitfalls in using automatic memoization. In *Proceedings of the 6th International Symposium on Artificial Intelligence*.

[8] Thomas Henderson, Sumit Roy, Sally Floyd, and George Riley. 2006. ns-3 project goals. In *Proceedings of the 1st Workshop on ns-2: The IP Network Simulator*.

[9] Ralf Hinze. 2000. Memo functions, polytypically! In *Proceedings of the 2nd Workshop on Generic Programming*. 17–32.

[10] Maria Hybinette and Richard Fujimoto. 1997. Cloning: A novel method for interactive parallel simulation. In *Proceedings of the 29th Winter Simulation Conference*. 444–451.

[11] Maria Hybinette and Richard Fujimoto. 2001. Cloning parallel simulations. *ACM Trans. Model. Comput. Simul.* 11, 4 (2001), 378–407.

[12] ISO. 2014. *ISO/IEC 14882:2014 Information Technology — Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland. 1358 pages.

[13] Kazutaka Kamimura, Ryosuke Oda, Tatsuhiro Yamada, Tomoaki Tsumura, Hiroshi Matsuo, and Yasuhiko Nakashima. 2012. A speed-up technique for an auto-memoization processor by reusing partial results of instruction regions. In *Proceedings of the 3rd International Conference on Networking and Computing*. 49–57.

[14] William Landi. 1992. Undecidability of static analysis. *ACM Lett. Prog. Lang. Syst.* 1, 4 (1992), 323–337.

[15] Pierre L'Ecuyer. 1999. Good parameters and implementations for combined multiple recursive random number generators. *Operat. Res.* 47, 1 (1999), 159–164.
[16] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30.
[17] James Mayfield, Tim Finin, and Marty Hall. 1995. Using automatic memoization as a software engineering tool in real-world AI systems. In *Proceedings of the 11th Conference on Artificial Intelligence for Applications*. 87–93.
[18] Paul McNamee and Marty Hall. 1998. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Not.* 33, 8 (1998), 17–22.
[19] Donald Michie. 1967. *Memo Functions: A Language Feature with "Rote-Learning" Properties*. Technical Report. Edinburgh University, Dept. of Machine Intelligence and Perception.
[20] Donald Michie. 1968. Memo functions and machine learning. *Nature* 218, 5136 (1968), 19–22.
[21] Jack Mostow and Donald Cohen. 1985. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. 165–172.
[22] Peter Norvig. 1991. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics* 17, 1 (1991), 91–98.
[23] Robin Popplestone. 1967. *Memo Functions and the POP-2 Language*. Technical Report. Edinburgh University, Dept. of Machine Intelligence and Perception.
[24] Robert Rönngren, Michael Liljenstam, and Johan Montagnat. 1996. Transparent incremental state saving in time warp PDES. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. 70–77.
[25] Markus Schordan, Thomas Oppelstrup, David Jefferson, and Peter Barnes. 2016. Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In *Proceedings of the 4th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 111–122.
[26] Christoph Sommer. 2017. Veins. (23 June 2017). veins.car2x.org, retrieved Aug 17, 2017.
[27] Christoph Sommer, Reinhard German, and Falko Dressler. 2011. Bidirectionally coupled network and road traffic simulation for improved IVC analysis. *IEEE Trans. Mobile Comput.* 10, 1 (2011), 3–15.
[28] Mirko Stoffers, Ralf Bettermann, and Klaus Wehrle. 2017. Automated memoization: Automatically identifying memoization units in simulation parameter studies. In *Proceedings of the 21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*.
[29] Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2016. Automated memoization for parameter studies implemented in impure languages. In *Proceedings of the 4th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 221–232.
[30] Tomoaki Tsumura, Ikuma Suzuki, Yasuki Ikeuchi, Hiroshi Matsuo, Hiroshi Nakashima, and Yasuhiko Nakashima. 2007. Design and evaluation of an auto-memoization processor. In *Proceedings of the 25th International Multi-Conference on Parallel and Distributed Computing and Networks*. 230–235.
[31] Andràs Varga. 2001. The OMNeT++ discrete event simulation system. In *Proceedings of the 15th European Simulation MC*.
[32] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2010. Autonomic Log/restore for advanced optimistic simulation systems. In *Proceedings of the 18th Symposium on Modeling, Analysis and Simulation of Comp. and Telecommunication Systems*. 319–327.
[33] Cheng-Xiang Wang, Matthias Pätzold, and Qi Yao. 2007. Stochastic modeling and simulation of frequency-correlated wideband fading channels. *IEEE Trans. Vehic. Tech.* 56, 3 (2007), 1050–1063.
[34] Darrin West and Kiran Panesar. 1996. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. 78–85.
[35] Neng-Fa Zhou and Taisuke Sato. 2003. Efficient fixpoint computation in linear tabling. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. 275–283.

Received November 2016; revised August 2017; accepted February 2018

Online Appendix to: On Automated Memoization in the Field of Simulation Parameter Studies

MIRKO STOFFERS, DANIEL SCHEMMEL, OSCAR SORIA DUSTMANN, and KLAUS WEHRLE, Communication and Distributed Systems, RWTH Aachen University

A INPUT VECTOR COMPUTATION DETAILS

This appendix cites additional details on the Input Vector (IV) computation from our PADS paper, namely how the adapted dead code elimination works and how the Temporary Cache (TC) is implemented and helps detecting aliases.

A.1 Adapted Dead Code Elimination

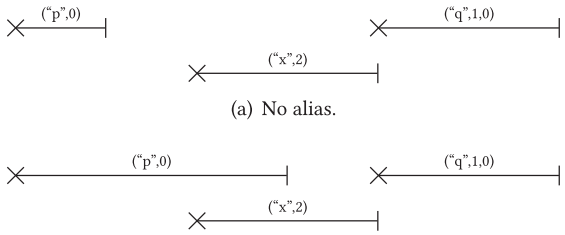
It is essential to avoid complex operations during IV reading. To this end, we use a transformation derived from standard dead code elimination, which we extend for this purpose. As opposed to simpler analyses, this is capable of efficiently dealing with complex IV calculations. One interesting class 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 that are no longer necessary are successively removed. For example, if a value `x` is stored in an exterior variable, which is never read afterwards, then 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 is not trivial. If the base pointer is interior, then 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 is insufficient 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 runtime conditions. A dynamic check could be performed, e.g., by determining whether the pointer points into the stack segment holding the local variables of the Memoization Unit (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.

1049-3301/2018/09-ART26 $15.00
https://doi.org/10.1145/3186316



(a) No alias.

(b) Overlap with the previous object.

Fig. 1. Alias detection: inserting ("x",2).

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. Note that this property can easily be checked statically. This constraint enables us to determine whether an object reached by $(X, \ldots)$ 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 $I = \{(X, \ldots) | X\text{read} \wedge X\text{not interior}\}$ and the output as $O = \{(X, \ldots) | X\text{written} \wedge X\text{not interior}\}$.

However, another requirement exists to ensure correctness: Since, in general, the same object may be reached via multiple different paths, 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 Temporary Cache (TC) discussed in the next section provides a way to detect aliasing and gracefully degrade to unmemoized execution in that case.

A.2 The Temporary Cache

At the most basic level, the Temporary Cache (TC) is a dictionary that maps memory addresses[1] 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, then an entry is immediately inserted into the TC, which will 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

[1]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.

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 Figure 1, 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.