Automated Memoization: Automatically Identifying Memoization Units in Simulation Parameter Studies

Mirko Stoffers, Ralf Bettermann, Klaus Wehrle
Communication and Distributed Systems
RWTH Aachen University
{stoffers, bettermann, wehrle}@comsys.rwth-aachen.de

***Abstract*—Simulations, and in particular large scale parameter studies, typically exhibit a considerable amount of redundancies. These redundancies can be avoided by memoization, a technique that stores and re-uses intermediate results. This requires a Memoization Unit (MU) to be identified first and then transformed. We have recently enabled the automation of the second step to also be applicable to impure computations, allowing it to become a valuable tool for the modeling and simulation domain. However, the first step still needs to be performed manually. Hence, the user needs to understand the model and the concept of memoization well enough to specify which computations to annotate for memoization.**

In this paper, we describe our approach to automatically identify memoization-worth computations. Input to this algorithm is an unmodified parameter study. After identifying the most promising memoization opportunities, we use the existing automated memoization tool to create a memoized parameter study, which can then be executed quickly.

Our evaluation shows that our automated approach is able to identify those MUs that previously had to be annotated manually. This identification takes less than 2 minutes for a case study that without memoization takes several hours.

I. Introduction

Computer simulations, and especially large scale parameter studies, are highly repetitive processes with a lot of redundant computations: the same code is executed on the same input, hence yielding the same result. Memoization allows to avoid such redundant computations by caching previously computed input-output pairs and applying the output directly if the same input re-occurs. Our recent advances [17] made automated memoization a valuable optimization technique for the modeling and simulation domain, reaping an optimization potential orthogonal to parallelization. Previously, memoization had to be applied manually in most common cases as automated memoization was only available for pure functions[1]. The lifting of this barrier now allows automated memoization to be applied in the modeling and simulation domain. A fully automatic memoization comprises two steps: 1) Promising code blocks have to be identified. 2) The code blocks have to be transformed into a memoized variant. In [17] we present our solution to the second step, which relies on the user to annotate a Memoization Unit (MU), i. e., a code block that should be memoized. In other words, the first step has to be performed manually, which is an open issue till today. In this paper, we focus exactly on this open issue, and present our approach to automatic identification of promising MUs, eventually allowing a fully automatic memoization.

Our approach is designed for large scale parameter studies since these comprise the highest degree of redundancy. Redundant computations can occur during the execution of one configuration, but also across configurations. The latter is particularly likely when a parameter is changed while the others are kept constant, which must always be done to investigate the impact of that parameter. We focus on parameter studies following the principle of full factorial design. This principle enables the user to make very precise statements about the influence of each parameter and combinations thereof, but often results in prohibitively long execution times. As a result, researchers tend to explore only a smaller subset, which poses the risk of false conclusions. On the other hand, the exploration of such a subset could already add enough entries to the Memoization Cache (MC) to execute the remaining configurations significantly faster. Hence, we hope that the availability of fully automatic memoization for full factorial design parameter studies can increase the willingness of researchers to apply this valuable design principle.

Our approach aims at identifying the most gainful MUs in a given parameter study. Since the true optimum can only be investigated by exactly measuring the performance of all opportunities, it is practically infeasible to do so. Instead, we design a heuristical approach that investigates the most promising opportunities and predicts their runtime. Hence, the major challenge we tackle is to design the heuristic in a way that it finds a close-to-optimal memoization opportunity, but at the same time performs this process quickly enough to achieve speedup in the final reckoning. In our experiments, we measure that time and compare the identified MU with the best opportunity that we figured out manually.

The remainder of this paper is structured as follows: In Sect. II we provide background information on the underlying memoizer as well as parameter study designs. After that, we analyze the problem in more detail (see Sect. III) before we discuss our solution (see Sect. IV). We then evaluate the overhead and performance in Sect. V. We discuss related work (see Sect. VI) before we conclude the paper (see Sect. VII).

[1]As we defined in [17]: "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."

978-1-5386-4028-0/17/$31.00

II. Background

In this section we introduce the underlying memoization approach, discuss common parameter study designs, and elaborate on the impact of memoization on the different principles.

A. Memoization

Memoization is a technique enabling the computer to "remember" previous computations. In other words, if the same code fragment is executed with the same input, the result can be applied immediately avoiding the re-computation. To develop a technique to automatically identify memoization-worth computations, we need a tool to eventually translate the computations into a memoized equivalent in order to predict the potential gain. Since many simulation models, including those written for the well-known open-source simulation frameworks ns-3 [7] and OMNeT++ [19], are implemented in programming languages such as C++ where impure computations are the prevalent paradigm, this memoizer must be capable of memoizing impure computations. To the best of our knowledge till today our recently developed memoizer [17] is the only one fulfilling this essential constraint.

We build upon the proof-of-concept implementation `memoize`, which uses the C++ frontend of the compiler Clang[2] and is publicly available[3]. It performs a source-to-source translation to convert a source file into a memoized version, which can be compiled into a binary by any C++14 compiler. To this end, the user must annotate a compound statement to be memoized (called the Memoization Unit (MU)). The tool then generates code that first performs only operations necessary to compute the input of the computation. Second, a lookup in the Memoization Cache (MC) is performed. If the input is not found, the original computation is executed and meanwhile the output is captured, such that input and output can be stored in the MC afterwards. On re-occurrence of the same input, this input-output pair will be found in the MC and the output is applied immediately. For this paper, we only modified the interface of `memoize`, i. e., the specification of the MUs, for easier communication with our identification tool, but left it otherwise unchanged.

B. Parameter Study Design

We base our approach to identify computations for memoization on parameter studies, a domain in which we expect many redundant computations. Jain [12] discusses methodologies for such designs and identifies simple design, fractional design, and full factorial design as most frequently used.

In simple design, a base configuration is chosen and only one parameter is changed at a time. This allows observing the influence of each factor separately, but the influence of factor combinations can not be discovered and interactions between factors are hidden. Hence, false conclusions might be drawn if factors interact. Simple design thus enables quick execution at the price of potentially questionable results.

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

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

The more statistically sound principle is full factorial design, which evaluates all possible combinations of parameters. This, however, results in $\prod_{i=1}^{n} k_i$ configurations for n parameters and k_i levels for parameter i, i. e., the number of configurations is exponential in the number of parameters. While this allows observing the influence of any combination of parameters, the exponential number of configurations results in long execution times, deterring many researchers from using it.

In fractional factorial design only a subset of all configurations is executed. This allows observing some parameter interactions, but not all, making it a compromise in both time consumption and statistical expressiveness.

We conclude that full factorial design generates the most reliable results, but takes most time as well. Hence, it is especially desirable to reduce the computation time of such a design. In particular, memoization is beneficial here: The execution of certain configurations is often skipped (by using fractional or simple design instead) since it is expected that no value is added by these configurations. If this is true, the additional runs comprise in fact a lot of redundancies – which perfectly aligns with the objective of memoization. However, with memoization this need not be assumed beforehand, but can be correctly evaluated by the memoized code at runtime. Hence, executing a full factorial design in a memoized manner might not take significantly longer than executing a fractional factorial design where only redundant configurations have been removed. However, the former guarantees that no significant computations have been removed while the latter yields false results if false assumptions have been made.

III. Problem Analysis

We aim for a tool to memoize a given parameter study. The problem we need to solve receives a simulation model and the configurations as input, and shall provide a memoized simulation model as output. The resulting model shall be memoized in a way as to yield the biggest possible benefit for the given configurations. Hence, we need to *predict* the benefit of different opportunities and choose the most promising.

To make such a prediction feasible, we need to make assumptions on the given parameter study. We identified full factorial design as most promising in Sect. II-B. Additionally, its structured construction supports performance predictions by analyzing only a small subset and extrapolating on the whole study that is later executed in a memoized way. We expect as input a parameter study given by the parameters and for each parameter the values to investigate. We then assume that all combinations shall be explored during the final execution of the parameter study. It is still possible to execute only a subset of the study (fractional factorial design), but the gain predicted by our performance analysis might not be achieved. Nevertheless, after applying memoization we would encourage users to try performing the remaining runs of a full factorial design as well, as chances are high that the results of the computationally expensive parts of the executions are already present in the MC, hence the additional time might be way shorter than expected. To state the problem more precisely,

we introduce a simple formalism defining a simulation, a parameter study, and the problem to solve.

a) Formal Problem Description: A simulation model is essentially just a function with a set of parameters as input and a set of results as output. With $\mathcal{F}$ being the set of all functions, we define a simulation instance as a tuple containing everything necessary to describe a certain simulative experiment:

Definition 1 (**Simulation Instance**)**.** We define a *simulation instance* s as $s = (f, c)$ with $f \in \mathcal{F}$ and $c \in \text{Dom}(f)$. The *result* of s is $f(c)$. We call c the *configuration* of s. We use $\mathcal{S}$ to denote the set of all simulation instances, i. e., $\mathcal{S} := \{(f, c) \mid f \in \mathcal{F}, c \in \text{Dom}(f)\}$.

We can define a parameter study as a tuple consisting of the simulation model (a function in $\mathcal{F}$) and a set of configurations. Important to note is that the set of configurations need not be ordered, allowing to execute the simulation instances in any order. Consequently, we define the result of a parameter study as the relation between an input and its corresponding output rather than just a set of outputs.

Definition 2 (**Parameter Study**)**.** We define a *parameter study* p as $p = (f, C)$ with $f \in \mathcal{F}$ and $C \subseteq \text{Dom}(f)$. The *result* of p is $\{(c, r) \mid c \in C, r = f(c)\}$. The *size* of p is the number of configurations $|C|$. By $\mathcal{P}$ we denote the set of all parameter studies, i. e., $\mathcal{P} := \{(f, C) \mid f \in \mathcal{F}, C \subseteq \text{Dom}(f)\}$.

We do not particularly consider experiments that only differ in the Random Number Generator (RNG) seeds. However, in fact our notation covers this by simply considering the seed part of the configuration. A ten-times repeated experiment with otherwise identical parameters would hence be ten configurations only differing in the RNG seed parameter.

These definitions describe the functionality and results of parameter studies, but ignore runtime complexities of particular implementations. For memoization speedup predictions we need to extend our notation to include the time component of programs that use memoization and programs that don't.

A *program* is characterized by the function it computes and the (average) time it takes (on a given hardware) to execute the program on a particular input. Consequently, with $\mathcal{T}$ being the time domain (i. e., the set of durations measured in wall clock time) we define a program as follows:

Definition 3 (**Program**)**.** A program q is a tuple $q = (f, t)$ with $f \in \mathcal{F}$ and $t : \text{Dom}(f) \to \mathcal{T}$. The function t maps each potential input of f, i. e., each $c \in \text{Dom}(f)$, to the expected execution time in $\mathcal{T}$.

Memoization changes the runtime characteristics, which then depend as well on the initial state of the MC. We hence define a memoized program with the set $\mathcal{M}$ of possible MCs:

Definition 4 (**Memoized Program**)**.** A memoized program q is a tuple $q = (f, t, m)$ with $t : \mathcal{M} \times \text{Dom}(f) \to \mathcal{T}$, $f \in \mathcal{F}$, and $m : \mathcal{M} \times \text{Dom}(f) \to \mathcal{M}$. The function t maps each potential input MC combined with each input of f, i. e., each $(M, c) \in \mathcal{M} \times \text{Dom}(f)$, to the expected execution time in $\mathcal{T}$. The function m maps the initial MC state to its final state after execution of the given configuration.

Note that a non-memoized program can hence also be presented as a memoized one. The non-memoized program ignores the MC, hence the output MC of m is always identical to the corresponding input MC ($\forall c \in \text{Dom}(f) \forall M \in \mathcal{M} : m(M, c) = M$) and t does not depend on its first parameter ($\forall c \in \text{Dom}(f) \forall M_1 \in \mathcal{M} \forall M_2 \in \mathcal{M} : t(M_1, c) = t(M_2, c)$). We denote the set of memoized programs (which hence also includes non-memoized ones) by $\mathcal{Q}$. According to this notation the MC generated by subsequent execution of n memoized programs is described as $m(\dots m(m(\varnothing, c_0), c_1), \dots c_n)$ with $\varnothing$ being the empty MC. To facilitate reading we let this be equivalent to the shorthand notation $m(\varnothing, \{c_0, \dots, c_n\})$.

The runtime of a parameter study depends on the set of configurations and the program used to compute the results. If a memoized program is used, the runtime might additionally depend on the order in which the configurations are executed since the resulting MC of a configuration is input to the next configuration. Hence, we denote the total runtime as a function $T : \mathcal{P} \times \mathcal{Q} \times \lessdot \to \mathcal{T}$ with $\lessdot$ being the set of strict total orders. The runtime is then calculated as:

$$T(p = (f, C), q = (f, t, m), <) = \sum_{c \in C} t(c, m(\varnothing, \{c' \mid c' < c\}))$$

b) Problem Statement: Given is a parameter study $p = (f, C)$ and a non-memoized program $q_0 = (f, t_0)$. A (not necessarily finite) set $Q = \{q_1, q_2, ...\}$ ($q_i = (f, t_i, m_i)$) of functionally equivalent memoized versions of q_0 exists, which can be generated by a tool for automated memoization with annotated MUs. The goal is to minimize the total runtime of p by choosing the best program from $Q \cup \{q_0\}$, i. e., select a $q \in Q \cup \{q_0\}$ such that $\nexists q' \in Q \cup \{q_0\} : T(p, q', <) < T(p, q, <)$. Note that this goal is rather a theoretical description of the optimum that can in practice never be reached. In fact it is neither feasible to select the optimal program from an unknown, potentially infinite set nor to prove non-existence of a better one. In this paper, we reduce the search space to a number of programs that are created by letting the tool we discussed in [17] memoize a block of the unmemoized, user implemented source program. We then predict the runtime of those opportunities and select the most promising one.

c) Challenges: Memoization yields the biggest benefit if the MU is computationally complex and repeated frequently with the same input. The identification of complex and frequently re-used computations can be approached by static or dynamic analysis: We argue that the former has very limited opportunities to inspect those properties that depend on the parameters of the parameter study as – even when provided at compile time – computations basing on these parameters quickly render the analysis infeasible. This impedes predicting redundancies. Dynamic analysis is feasible, but we need to ensure that the fraction of the parameter study we execute during the analysis is small enough to achieve an overall gain.

With this constraint an exact calculation of the total runtime is impossible. In particular, if different configurations exhibit

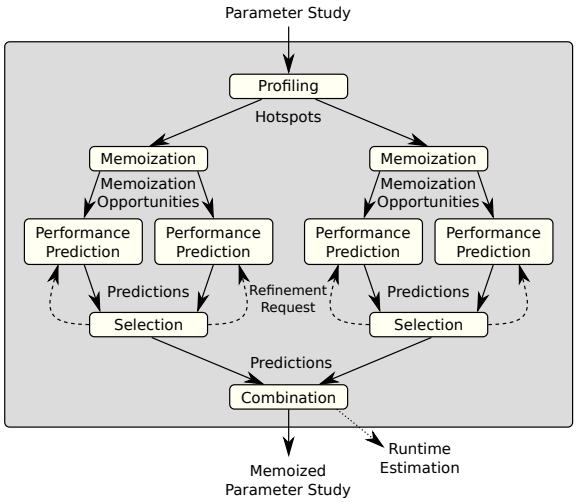


Fig. 1. General approach: Profiling identifies potentially optimizable hotspots. Different memoization opportunities are investigated and their performance is predicted. The following selection potentially requests refinements of the predictions. Selected optimizations of different hotspots are combined. Result is an optimized parameter study, and, as a side product, a runtime estimation as detailed as it was generated during the prediction.

largely deviating runtimes even before memoization is applied, it would be necessary to run a large amount of configurations for accurate runtime calculations. Hence, we need to predict the runtime by a heuristic based on a smaller number of configurations. To yield the best memoization decision, we do not target a maximum absolute prediction accuracy. Nevertheless, if program q_1 is faster than q_2, the heuristic should predict a runtime of q_1 that is shorter than the runtime of q_2. The challenge is thus to design a heuristic that 1) works only on a particularly small subset of the parameter study, but 2) yields a runtime prediction that can be used to select the most promising code block for memoization. Obviously, we need to trade dynamic analysis runtime for prediction accuracy and need to find a tradeoff that minimizes the total runtime of analysis plus execution of the parameter study.

IV. Memoization Unit Selection

In this section we present our approach to selecting code blocks that are promising for memoization. We outline the general approach before we introduce each step in more detail.

A. General Approach

The general approach is outlined in Fig. 1. Input is a parameter study, i. e., an implemented simulation model and the configurations to be explored. We analyze the input for its most computationally complex computations as optimizing these hotspots carries the greatest potential. Around each hotspot we choose multiple memoization opportunities and perform the automated memoization by `memoize` (s. Sect. II-A), resulting in several memoized alternatives. The core element of our approach then predicts the performance of each opportunity. To save time we allow this prediction to be incomplete, for instance, by only providing a lower bound of the runtime prediction. In a next step, we select the most promising opportunity. Should this be impossible due to the incompleteness of

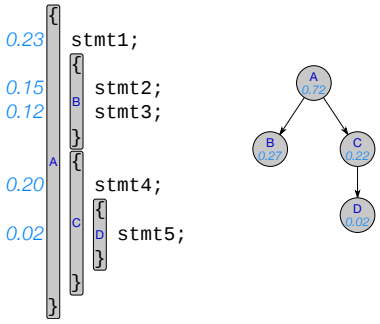


Fig. 2. Simple example: Code consisting of five statements (`stmt1`-`stmt5`) in four blocks (A-D). Profiling data (fraction of runtime spent on each statement) displayed on the left. The tree on the right shows structured and aggregated profiling data inferred by our tool.

the prediction, the prediction is refined. After the selection the best opportunities for each hotspot are combined, yielding the most promising memoized parameter study. As a side product, we retrieve a runtime estimation, which, however, is only as accurate as necessary in the realm of the main goal. In the following, we discuss each step in more detail.

B. Profiling

In the spirit of Amdahl's Law [1] we need to profile the input to determine which parts are most promising for optimization. Statistical profiling is promising for our use case since it is typically faster [2] and sufficiently accurate. We integrate the commonly available profiler OProfile [4] in our proof-of-concept implementation, but allow to use any profiler whose output can be parsed in a way that it provides pairs of code locations and their frequency of occurrence.

More importantly, we need to select a fraction of the parameter study to profile. To this end, we have to assume a certain level of homogeneity of the parameter study: If every configuration results in different hotspots, no memoization opportunity fits all of them. However, we still aim to minimize the probability to select a corner case configurations (e. g., all parameters set to zero) whose hotspots are not representative. Since this might often be the first or last configuration, deterministically choosing either of them is not advisable. Instead, we select a configuration randomly to maximize the probability not to choose a corner case. Additionally, if the overall result is not satisfying because a corner case was accidentally selected, this allows to run our tool again, which will then likely select a different configuration.

Experience shows that for most simulation models it is sufficient to profile only a small fraction of the program itself. To speed up the selection process, we run the model for only 2 s by default. However, if a model maintains an extensive initialization phase, this value can be adjusted to profile a sufficient fraction of the actual execution.

The result of the profiling is the fraction of runtime consumed by each statement in the program (see Fig. 2). The fractions can be summed up to compute the percentage of runtime spent in a compound statement up to the block that constitutes the body of a non-inlined function. Since non-inlined function calls can not be memoized [17], this

constitutes as well the biggest compound statement that can be memoized. We rank the functions by the percentage of runtime spent in their body and include the most expensive hotspots until their accumulated runtime is greater than 50 %. For each hotspot we then have the runtime spent in this block as well as the runtimes for each sub-block.

C. Memoization

Input to the memoization step is a single hotspot (which is a function body) annotated with the runtime of each compound statement (cf. Fig. 2 (b)). In general, each node is a memoization opportunity, i. e., each compound statement can constitute an MU. However, we can already prune all compound statements whose complexity is less than the complexity of the most complex function that is not considered a hotspot, since these blocks are less promising than that function. For a threshold of 25 % in Fig. 2, C and its child would be pruned.

For each of the remaining blocks we provide the code along with the information which compound statements are potential MUs to `memoize` and retrieve a memoized variant for each memoization opportunity. All these variants (as well as the original program) are input to the next step.

D. Performance Prediction

Before the runtime of the parameter study can be predicted, the influence of each parameter on the redundancy of the potential MUs has to be investigated. When not changing any parameter between two runs, the entire execution is identical. When a parameter is changed, all computations independent of this parameter (and any derived values), are repeated. When two or more parameters are changed, those computations are repeated that depend on neither of the parameters.

For the performance prediction, we decompose the runtime of a configuration (s. Def. 4) into several components. For a given (memoized) program $q = (f, t, m)$ and a configuration $c = (v_1, \ldots, v_k) \in \mathrm{Dom}(f)$, we decompose $t(c)$:

$$t(v_1, \ldots, v_k) = \sum_{I \subseteq \{1,\ldots,k\}} t_I((v_i)_{i \in I})$$

We call t_I a *partial runtime* and consider it being the runtime caused by performing computations whose input changes if any of the parameters $(v_i)_{i \in I}$ is changed, but there is no subset $J \subseteq I$ such that the same holds for J (in which case it would be reflected by t_J), e. g., the time spent in the pure function called by `foo(v4,v2)` contributes to $t_{\{2,4\}}$. More precisely, t_I reflects the amount of computations that the given (memoized) program q needs to re-perform when any of the parameters in I is changed, either because the modified parameters change the input to the computations performed, or because the MU has been chosen in a way not avoiding the redundant computation. Consequently, $t_\varnothing$ reflects the time that can be avoided even if all parameters are changed, $t_{\{1,\ldots,k\}}$ reflects the time that cannot be avoided even if all parameters are kept constant. Since the memoization overhead can never be avoided, it is nicely covered in $t_{\{1,\ldots,k\}}$, so it need not be considered explicitly in the following steps. Similarly, a non-memoized program would not avoid any computations, resulting in $t = t_{\{1,\ldots,k\}}$ and $t_I = 0 \ \forall I \neq \{1, \ldots, k\}$.

We must note that this decomposition is an abstraction as it suggests t_I was independent of the computations reflected by t_J ($J \neq I$), which is not entirely true for reasons such as caching or branch prediction. The decomposition is still accurate enough for a reasonable runtime prediction.

Unfortunately, a large amount of parameters inevitably results in a prohibitively large amount of partial runtimes, namely the size of the power set. To solve this issue, we reduce the parameter space that we need to explore:

a) Parameter Space Reduction: If a potential MU is influenced by many parameters, the chance to find a lot of redundant computations is low since the computations do not re-occur if just any parameter is changed. Further exploring such a memoization option is not promising. Well-suited candidates for memoization, on the other hand, are those MUs whose input is influenced only by a few parameters. Hence, in a first step we need to determine the influential parameters.

To this end we execute two configurations: $c_1 = (v_1, \ldots, v_i, \ldots, v_k)$ and $c_2 = (v_1, \ldots, v_i', \ldots, v_k)$ for random parameters, but ensuring $v_i \neq v_i'$. If a cache miss occurs during execution of c_2, or an entry generated by c_1 is not re-used by c_2, we know that the MU input is influenced by parameter i. We denote i as a *relevant* parameter. Otherwise parameter i did not influence the input of the MU and we treat it as *irrelevant*. We must note that we might erroneously treat a parameter as irrelevant when it influences the MU computation only under certain conditions. In this case we would overestimate the potential of this MU, which is, however, the nature of a heuristic that it cannot always be correct.

For a parameter study with k parameters in total, and k_r relevant parameters this step adds $k + 1$ computations, but reduces the number of partial runtimes to compute in the next step from 2^k to 2^{k_r}. We impose a fixed upper limit for k_r and deem the choice of the MU inappropriate if the limit is exceeded. Our experiments showed that 3 is a good value for this, limiting the number of partial runtimes to 8.

b) Estimation of Partial Runtimes: If a parameter $i \in I$ is irrelevant, $t_I = 0$ since either no computations have to be re-performed if i changes or the computations must also be re-performed if a parameter $j \in J = I \backslash \{i\}$ changes. With $I_\mathrm{R} \subseteq \{1, \ldots, k\}$ being the set of relevant parameters we can express the reduced runtime decomposition as follows:

$$t(v_1, \ldots, v_k) = \sum_{I \subseteq I_\mathrm{R}} t_I((v_i)_{i \in I})$$

We model the memoization overhead in t_{I_R}, now. The formula consists of $2^{|I_\mathrm{R}|}$ components, i. e., less or equal 8 for $|I_\mathrm{R}| \leqslant 3$. By executing pairs of configurations (c_1, c_2), we can derive concrete instances of this formula. The left hand side of the formula is always the runtime of c_2. Some of the summands on the right hand side are 0 since memoization avoids the corresponding computations (and the overhead is reflected in t_{I_R}). For instance, $t_{\{1\}}$ is 0 if parameter 1 is kept constant from c_1 to c_2 and $t_{\{1\}}$, $t_{\{3\}}$, and $t_{\{1,3\}}$ are 0 if parameters

1 and 3 are kept constant. The only exception to this rule is t_{I_R}, which is never set to 0, as there will always be computations that cannot be avoided, namely the memoization overhead and the computations outside the MU. $2^{|I_R|}$ pairs of configurations generate $2^{|I_R|}$ equations with $2^{|I_R|}$ unknowns. Choosing the configurations such that the set of changed parameters is different for each equation yields all pair-wise linearly independent equations, hence solving the equation system computes the values of the unknowns $t_\varnothing$, $t_{\{1\}}$, ..., t_{I_R}.

c) Estimation of Total Runtime: To predict the total runtime of the parameter study we recompose the partial runtimes to deliver the runtime of each configuration. The first configuration cannot re-use results computed in previous configurations, hence its runtime is $\sum_{I \subseteq I_R} t_I((v_i)_{i \in I})$. Setting times of avoidable computation to 0 allows us to compute a prediction for the execution of each configuration under a certain order. In fact, the order influences the runtime of a configuration, but not the overall runtime as swapping two configurations just moves the effort between them. Hence, summing up the predicted runtime of each configuration yields a runtime prediction for the entire parameter study that is memoized in the provided way.

d) Optimizations: The following issue has been ignored so far: Executing certain configurations during the prediction might take prohibitively long if the MU selection was poor. Similarly, the unmemoized execution might take prohibitively long as well (cf. the Fibonacci example in [17]).

Hence, we designed the dynamic analysis to allow stopping executions. Since we have no baseline to compare to beforehand, we use the available CPU cores and start multiple jobs at the same time. Different memoization opportunities (randomly chosen if the set to explore is greater than the number of CPU cores) plus unmemoized code are launched concurrently in the same configuration. After the fastest finished, we allow a certain amount of extra time for the others to complete since stopping immediately would be too restrictive due to possible fluctuation in the runtime. Hence, for some opportunities we might only retrieve a lower bound for the runtime.

If we need to cancel an execution during the parameter space reduction step, we deem the parameter having *uncertain* influence. We might be able to propagate the influence along the syntax tree: Relevant parameters are relevant for the parents, irrelevant parameters are irrelevant for the children. We treat parameters that still have uncertain influence as relevant, first. If this exceeds the threshold of 3 relevant parameters, we proceed with the next step, but execute only 8 configurations. Hence, for the runtimes depending on some of the parameters with uncertain influence we choose a lower bound of 0. We then determine a lower bound of the total runtime and provide this to the selection process described in the next section.

If we need to cancel an execution during the estimation of partial runtimes, we retrieve an inequality rather than an equation. Treating such an inequality like an equation, allows us to compute a lower bound for the runtime.

Finally, for every memoization opportunity, we retrieve either a runtime prediction or a lower bound thereof.

E. Selection and Combination

From the runtime predictions and lower bounds we select the numerically smallest value. If this is a runtime prediction, the corresponding memoization opportunity is the most promising one and is hence selected. If the value is only a lower bound, control needs to be passed back to the performance prediction step with more permissible runtime. This results either in a numerically greater lower bound or a precise result. We then repeat the selection step. We finally retrieve a runtime prediction that is provided as an exact value rather than a lower bound. However, in general we expect this procedure to be only a backup strategy since typically the need to cancel an execution due to a prohibitively long runtime indicates an unlikely candidate.

Input to the final combination process are the MUs chosen by the selection process for each hotspot and the corresponding runtime prediction. Since the selected MUs are found in different hotspots, they do not overlap and can all be memoized. The overall runtime estimation can be computed by subtracting the time saved by each MU from the runtime prediction of the original parameter study. We finally create a memoized program where each of the selected MUs is memoized.

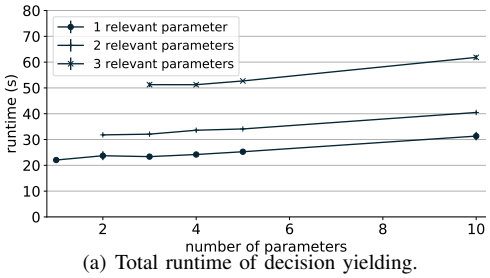
V. Evaluation

The goal of our evaluation is two-fold: First, we aim to provide general estimations on the accuracy of our predictions and its computation time. Second, we determine whether our tool is in practice able to choose the best memoization option. For the first, we designed a synthetic benchmark with varying number of parameters for which we can easily determine the runtime characteristics manually to compare it with the predictions (see Sect. V-B). For the second, we use the benchmarks we discussed in [17] where we had identified the most promising code blocks manually, such that we can determine if the automatic identification introduced in this paper comes to the same conclusion. We describe our results on Fibonacci number computation in Sect. V-C and the Orthogonal Frequency-Division Multiplexing (OFDM) fading model in Sect. V-D. Prior to that we introduce our general evaluation methodology in Sect. V-A.

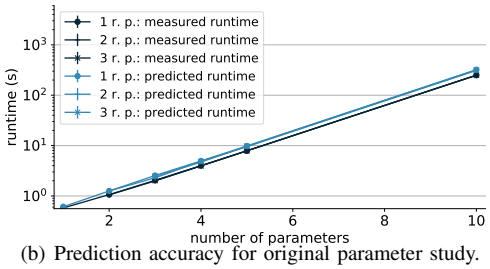
A. Evaluation Methodology

For each benchmark, we create a simulation model for OMNeT++ 5.0. We describe a parameter study in the OMNeT++ initialization file as discussed in [18, Section 10.4 "Parameter Studies"]. Hence, our tool can use OMNeT++ to retrieve all necessary information about the parameter study to optimize. For the memoization transformation we use version 20161220 of `memoize` and as a compiler we use Clang 3.9.

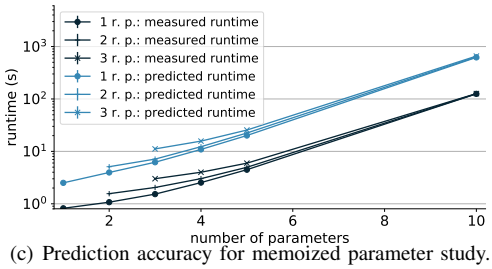
All experiments are performed on a compute server with 2 Intel Xeon E5-2643 v4 CPUs (2×6 cores, hyper-threading disabled) with 256 GB RAM. We repeat every experiment at least 10 times and show means and 99 % confidence intervals in every plot.



(a) Total runtime of decision yielding.



(b) Prediction accuracy for original parameter study.



(c) Prediction accuracy for memoized parameter study.

Fig. 3. Synthetic benchmark: heuristic runtime and prediction accuracy.

B. Synthetic Benchmark

The main influence factors of the time and accuracy of our heuristic are the number of parameters and the number of relevant (s. Sect. IV-D) parameters. To investigate the scalability of our approach, we hence create a benchmark allowing us to vary both. Our simulation model consists only of a single event handler that purposefully performs some useless calculations and takes about 0.3 s. The code is designed in a way that a memoized version performs at about the same order of magnitude. We vary the number of parameters from 1 to 10, i. e., the parameter studies consist of up to 1024 configurations. We vary the number of relevant parameters from 1 to 3 and perform experiments with all possible combinations.

Fig. 3 depicts the results. Fig. 3(a) shows the runtime of our tool. We nicely observe the runtime increasing linearly in the number of parameters although the size of the parameter study increases exponentially. With varying number of *relevant* parameters, we observe the expected exponential increase, which confirms that this number need be small. Remember that memoization itself can as well only be beneficial if there are enough irrelevant parameters, such that redundancies actually occur inside the MU. For a large amount of reasonable inputs we can hence predict the runtime sufficiently fast.

Predicted and true runtimes are shown in Fig. 3(b) for the original, in Fig. 3(c) for the memoized study. We clearly observe the expected exponential runtime of the entire study. Our heuristic always overestimates the runtimes. We attribute this to the fact that for the prediction multiple CPU cores are used while for computing the actual runtime we only use a single core. This can hence be clocked higher and has unshared access to L3 cache and memory bus. The increased cache capacity especially helps the memoized version. Nevertheless, we observe that the general course and the differences between the configurations is correctly reflected. Hence, relative statements – as required – can be made accurately.

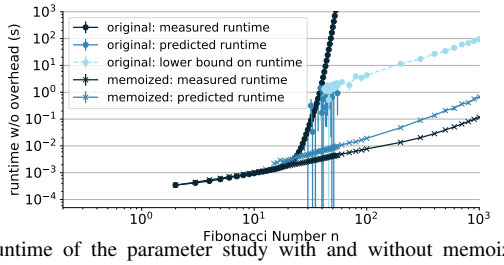
C. Fibonacci

The Fibonacci implementation discussed in [17] computes the nth Fibonacci Number in a standalone program. This is an interesting benchmark for this paper since it depends on the input parameter whether the computation is memoization-worth or not. To evaluate our approach we need to wrap the computation into a simulation model and a parameter study. Our model consists of a single module with a single event handler that computes the sum of Fibonacci numbers up to F_i. This i is the only parameter of the study, which consists of n configurations with $i = 1..n$. In our experiments, we vary n from 2 to 1000 to identify how well our approach works with differently sized parameter studies with different complexities of the computations themselves. In total, we create 68 parameter studies and measure different time values and captured how our tool decided. There is only a single hotspot with a single memoization opportunity to be identified. Hence, the heuristic only decides whether memoization should be applied or not. However, for small Fibonacci numbers the computations are not complex enough to identify a hotspot. In this case the heuristic correctly concludes that memoization is not beneficial and emits the parameter study unmodified.

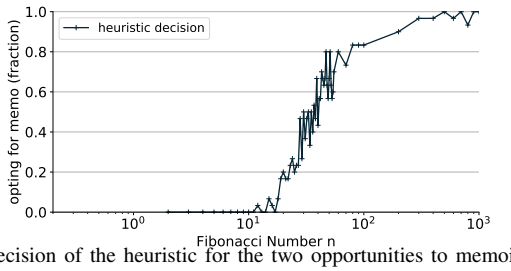
Fig. 4(a) compares predicted and measured runtimes of both the memoized and non-memoized version, excluding the OMNeT++ setup and teardown time. Up to about F_{25} no memoizable hotspot is identified, thus no predictions are available. Although the plot suggests memoization should be applied from F_{18} on, it should be noted that for such simple computations the OMNeT++ overhead dominates the overall runtime, such that the optimization would not be perceivable.

From F_{25} to about F_{50}, predictions are computed but are heavily fluctuating. This is caused by the nature of the parameter study: The execution time of a configuration heavily depends on the parameter. As discussed in Sect. III this renders a precise estimation almost impossible. Nevertheless, in most cases we correctly predict a higher runtime for the original version than for the memoized one. From about F_{50} on the likelihood of choosing a small Fibonacci number is small enough, such that in most cases the heuristic had to cancel the original implementation due to prohibitively long execution times and emitted only lower bounds on the runtime.

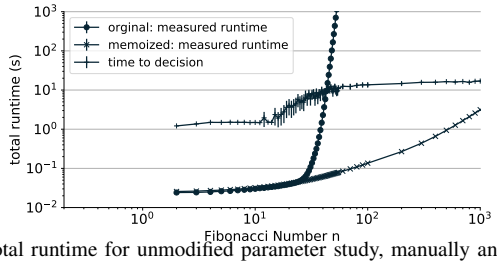
The actual runtimes follow our previous observations in [17] though the absolute runtime differs since we wrapped the computation into a parameter study and performed the experiments on different hardware. Our heuristic slightly overestimates the runtime of the memoized parameter study, but in general the prediction follows the true runtime adequately. All lower



(a) Runtime of the parameter study with and without memoization. Compares predictions with true results. Note that due to the extensive duration of the original implementation, for higher values our heuristic estimates only lower bounds on the runtime, which are sufficient to safely opt for memoization.



(b) Decision of the heuristic for the two opportunities to memoize or not to memoize based on 30 repetitions per experiment.



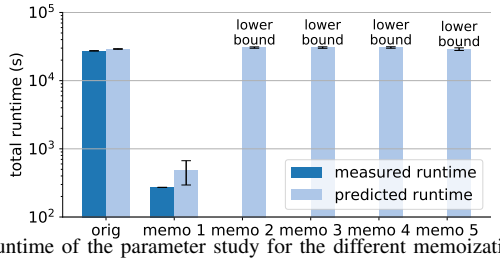
(c) Total runtime for unmodified parameter study, manually annotated MU with automated memoization, and time to select the best memoization opportunity.

Fig. 4. Evaluation results of the Fibonacci model.

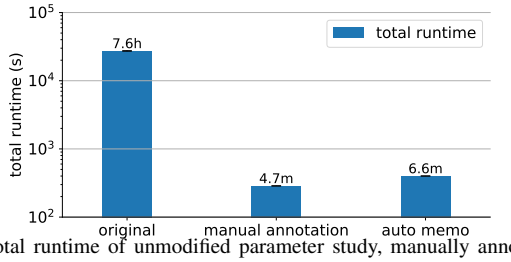
bounds for the predictions of the original implementation are (correctly) below the measured runtime of the original implementation. Additionally, they are sufficient to safely opt for memoization, hence no refinement is requested.

Fig. 4(b) shows the fraction of cases in which our tool opted for which alternative (original or memoization), based on the 30 repetitions we performed for each experiment. For small Fibonacci numbers our heuristic correctly identifies the original implementation as the better alternative. From F_{11} on, where both alternatives perform almost equally, it selects the original implementation in most cases. After that, the fraction of opts for memoization increases quickly, which correlates with the fact that now memoization becomes beneficial. However, even for larger values in some cases the heuristic still opts for the original implementation due to bad luck in the configuration selection. In this case, just re-executing our tool can help as the configurations are selected randomly.

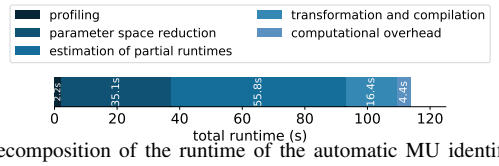
We assess the runtime of our tool in relation to the execution time of the model (s. Fig. 4(c)). For small Fibonacci numbers our tool adds considerable overhead compared to the actual execution time. This overhead increases up to 20 s, but stays constant at this level. Hence, from F_{40} on we are able to



(a) Runtime of the parameter study for the different memoization opportunities. Predicted and measured values. Measurements omitted for very long runs where prediction only provided a lower bound.



(b) Total runtime of unmodified parameter study, manually annotated MU with automated memoization, and fully automated memoization including time to select the best memoization opportunity.



(c) Decomposition of the runtime of the automatic MU identification into the different components.

Fig. 5. Evaluation results of the OFDM fading network simulation case study.

achieve overall improvement. Though our overhead is considerable in the investigated experiments, it is always less than 20 s. We argue that manual annotation cannot be performed within 20 s, hence we would outperform manual annotation if the annotation time was added to the entire runtime.

D. OFDM Fading Model

After manual MU identification we can reduce the runtime of an OFDM fading based parameter study by almost two orders of magnitude (s. [17]). We confirm this on the hardware described above (reducing the total runtime by a factor of 97 from 7.6 h to 4.7 min). Again, we evaluate whether we are able to perform the previously manual identification automatically.

We observe that our tool reliably detects the fading computation function as the most prevalent hotspot in the parameter study. This consists of 6 loops in total, out of which 4 are nested into each other. Since the function body itself can be memoized as well, this results in a total of 7 memoization opportunities. The 4 nested loops are of much greater complexity than the 2 others. For this reason as discussed in Sect. IV-C our tool prunes the 2 others, such that 5 opportunities remain to be investigated further. In the latter, in fact, a trade-off between input size and computational complexity could be expected, hence such deeper exploration actually makes sense.

Fig. 5(a) depicts our predictions. For opportunity 1 (entire function body) the heuristic computes a significant gain while for the others it predicts a performance decrease. The runtime is long enough to only generate a lower bound, which is safely

used to justify that memoization opportunity 1 is faster. We confirmed that the outer-most loop (memoization opportunity 2) in fact generates a large amount of intermediate results that are aggregated by one of the other loops later on. Additionally, the outer-most loop relies on further, previously generated, intermediate results. Hence, input and output of opportunities 2 to 5 is huge, resulting in prohibitive memoization overhead, greater than the complexity of the original fading computation.

For the original implementation the prediction is pretty close to the actual runtime. For the first memoization alternative the prediction overestimates the runtime and we observe quite a significant variance. In fact, not all configurations exhibit similar runtimes in the original implementation, such that the prediction varies with the configuration selected. However, since for all alternatives the same configurations are selected, this introduces inaccuracies in the absolute prediction, but not in the difference between predictions. Since the heuristic needs to be only good enough to generate results that are accurate relative to each other, the predictions are appropriate.

Due to the huge expected runtime of the remaining memoization alternatives we did not measure the exact runtime required. However, we did confirm for single runs that in fact the memoized version takes significantly longer than the original, even if the MC already contains the results required. Hence, the general conclusion that the runtime increases by applying one of these opportunities is correct.

We can hence conclude that our tool – correctly – identifies memoization alternative 1 as most promising.

Fig. 5(b) shows the performance achieved by the 3 options available now: 1) no memoization, 2) manual annotation of memoization opportunity 1 with automated memoization, and 3) automatic identification and memoization.

We observe the massive memoization gain already reported in [17] for both memoization approaches. However, while the manually annotated execution completes within less than 5 minutes, our approach takes an additional 114 s to identify the best MU. We argue, though, that our approach is much faster in identifying the MU than most users would be and conclude that fully automatic memoization is feasible.

Nevertheless, we further analyze the runtime our tool needs to improve our understanding of how the runtime is composed. To this end, we decompose the runtime into the several steps performed by our tool (see Fig. 5(c)). During the prediction, several configurations have to be investigated. This takes place in the steps Profiling, Parameter Space Reduction, and Estimation of Partial Runtimes. Since we execute only a single configuration partially for profiling, this step completes quite quickly. Most time is spent in the following two steps. This is expected since it is the most complex task to identify which parameters influence the runtime and how.

The alternatives are memoized and compiled, which as well takes a considerable amount of time. Finally, the plot depicts the time consumed by our heuristic (solving linear equation systems, etc.), which is relatively low. We conclude that, should optimization be necessary, the execution during the Estimation of Partial Runtimes would be the first target. However, we argue that the total overhead is quite low, compared to the gain achieved by avoiding the manual effort.

E. Parameterization

As discussed in Sect. IV, our approach allows to adjust a small set of parameters. Since the models evaluated coped fine with the default values, we didn't change any of those parameters. However, it is still important to note that the performance will decrease if, for example, a model maintains an extensive initialization phase, such that 2 s of profiling only profile the setup. In this case, this parameter need be adjusted. Similarly, it might be necessary to increase the maximum number of relevant parameters if no suitable MU is found.

F. Conclusion

Our evaluation shows that although the absolute runtime estimations are not always accurate, comparing two estimations still results in correct decisions. We are hence able to identify the most gainful MUs automatically, which previously had to be done manually. To this end, our heuristic needs less than 2 minutes in all experiments conducted. Hence, our heuristic is able to perform educated memoization decisions in very short time, eliminating the need to perform this effort manually.

VI. Related Work

Since automated memoization was not practically feasible until 2016, the urge for techniques to automatically identify promising computations for such a tool was not provided as well. To the best of our knowledge no approach has been developed so far to tackle this issue. However, the central component of our work is the runtime prediction of the parameter study. Software performance prediction has in fact been studied thoroughly in the past and we discuss these efforts in the following. Furthermore, there are alternative approaches to avoid redundancies that do not need the identification step. We outline these approaches as well.

a) Performance Prediction: In general, selecting the best memoization opportunity is a problem of algorithm selection. Algorithm selection has been first discussed by Rice in [15]. Kotthoff et al. [14] discuss a number of recent research efforts to select the best algorithm from a given portfolio by machine learning. In contrast to our situation, such portfolios are already limited to a reasonable number of algorithms. Hence, the input is comparable to the input of our performance prediction step discussed in Sect. IV-D. Generic algorithm selection solutions such as [3], [8], [9], [16] use different machine learning techniques to predict the value of the cost function (in our case the cost function would measure overall runtime). While such an approach might be applicable here as well, it would completely ignore the characteristics of parameter studies. As Rice already stated in 1976, proper algorithm selection "will always require exploitation of the specific nature of the situation at hand" [15]. In this vein we decided to tailor our performance prediction approach specifically for parameter studies rather than using a machine learning based approach off the shelf.

Other performance prediction techniques build upon simulation themselves. In this context, the input simulation model is considered just a piece of software as if it was not a simulation model itself. Prior to the prediction the code of the input software has to be transformed into a simulation model, i. e., just like with any simulation the modeling has to be done first. This model can then be executed to predict the software's performance on a given input. Note that in our case $|C|$ (number of configurations in the original study) simulation runs had to performed. A modeling method that needs no further manual effort is to compile the code and feed it into a cycle-accurate simulator such as Manifold [20], which allows to predict the performance. However, since cycle-accurate simulation emulates a single CPU cycle by multiple instructions, if only the runtime is of interest (like in our use case), just directly executing the code would always be faster.

Other approaches to simulative performance prediction [6], [13] perform static code analysis to automatically generate a way more abstract model, allowing to expect a speedup in the execution time. Nevertheless, this still requires the execution of $|C|$ simulation experiments, such that the runtime is still exponential in the number of parameters. Hence, an approach to reduce the parameter space and predict runtime based on this like our approach would still be necessary, though a combination might be an interesting option to explore.

b) Redundancy Avoidance: To avoid redundancies in parameter studies another technique worth mentioning is Simulation Cloning [10], [11]. This does not need the identification of promising computations as it starts off with only one simulation and "clones" any affected "virtual logical process" once its state is influenced by any of the parameters that are to be changed during the parameter study. Hence, only redundancies up to the first branching point can be avoided. Updateable Simulations [5] can be more effective in avoiding redundancies, but require manual implementation of update functions. In conclusion, automatic identification of promising computations for memoization is necessary to realize redundancy avoidance after the first branching point automatically.

Orthogonally to our work the memoizer itself can be optimized as well (e. g., by reducing overhead or increasing the applicability in multi-threaded contexts). This would of course result in improved performance. Hence, it would potentially change the results of our decision making process, but we argue that our methodology can still be applied in principle since the memoization tool itself is interchangeable.

VII. Conclusion

This paper introduces our approach to automatically identify computations well-suited for memoization. Motivated by Amdahl's Law [1] we first identify the most complex computations in the model. We then explore different memoization opportunities for those computations and heuristically select the most promising ones. We build upon our previously developed automated memoization tool [17] to transform the original implementation into a memoized one. Our evaluation shows that we are able to find the same MUs automatically that we previously had to identify manually. While manual identification is time-consuming and requires domain knowledge, our tool can do so automatically within less than 2 minutes.

In future efforts the time to run the final, memoized parameter study, can be reduced: During execution of certain runs we build up MCs, which we discard afterwards. By keeping the cache entries the memoized execution could benefit from the first computation on. Additionally, once automated memoization has been adapted for safe use in multi-threaded contexts, our performance predictions should take into account that the memoized parameter study can be executed in parallel as well.

To conclude, in this paper we demonstrate that automated memoization can practically be used without the need to manually identify the promising calculations beforehand.

Acknowledgments

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

References

[1] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proc. 1967 AFIPS Spring Joint Com. Conf.*, 1967, pp. 483–485.

[2] J. Anderson, L. Berc, J. Dean *et al.*, "Continuous Profiling: Where Have All the Cycles Gone?" *ACM TOCS*, vol. 15, no. 4, pp. 357–390, 1997.

[3] T. Bartz-Beielstein and S. Markon, "Tuning Search Algorithms for Real-World Applications: A Regression Tree Based Approach," in *Proc. 2004 Congress on Evol. Computation*, vol. 1, 2004, pp. 1111–1118.

[4] W. Cohen, "Tuning Programs with OProfile," *Wide Open Magazine*, vol. 1, pp. 53–62, 2004.

[5] S. Ferenci, R. Fujimoto, M. Ammar *et al.*, "Updateable Simulation of Communication Networks," in *Proc. 16th PADS*, 2002, pp. 107–114.

[6] S. Hammond, G. Mudalige, J. Smith *et al.*, "WARPP: A Toolkit for Simulating High-performance Parallel Scientific Codes," in *Proc. 2nd Conf. on Sim. Tools and Techniques*, 2009, pp. 19:1–19:10.

[7] T. Henderson, M. Lacage, and G. Riley, "The ns–3 Network Simulator," June 2014, www.nsnam.org.

[8] L. Huang, J. Jia, B. Yu *et al.*, "Predicting Execution Time of Computer Programs using Sparse Polynomial Regression," in *Proc. 23rd Conf. on Adv. in Neural Inform. Proc. Sys.*, 2010, pp. 883–891.

[9] F. Hutter, Y. Hamadi, H. Hoos *et al.*, "Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms," in *Proc. 12th Int. Conf. on Princ. and Pract. of Const. Prog.*, 2006, pp. 213–228.

[10] M. Hybinette and R. Fujimoto, "Cloning: A Novel Method for Interactive Parallel Simulation," in *Proc. 29th W. Sim. Conf.*, 1997, pp. 444–451.

[11] ——, "Cloning Parallel Simulations," *ACM Tran. Modeling and Com. Sim.*, vol. 11, no. 4, pp. 378–407, 2001.

[12] R. Jain, *The Art of Computer Systems Performance Analysis*. Wiley, 91.

[13] S. Jarvis, D. Spooner, H. L. C. Keung *et al.*, "Performance prediction and its use in parallel and distributed computing systems," *Future Generation Com. Sys.*, vol. 22, no. 7, pp. 745–754, 2006.

[14] L. Kotthoff, I. Gent, and I. Miguel, "An Evaluation of Machine Learning in Algorithm Selection for Search Problems," *AI Communications*, vol. 25, no. 3, pp. 257–270, 2012.

[15] J. Rice, "The Algorithm Selection Problem," in *Advances in Computers*, 1976, vol. 15, pp. 65–118.

[16] K. Smith-Miles and J. van Hemert, "Discovering the Suitability of Optimisation Algorithms by Learning from Evolved Instances," *Annals of Mathematics and AI*, vol. 61, no. 2, pp. 87–104, 2011.

[17] M. Stoffers, D. Schemmel, O. Soria Dustmann, and K. Wehrle, "Automated Memoization for Parameter Studies Implemented in Impure Languages," in *Proc. 4th ACM SIGSIM PADS*, 2016, pp. 221–232.

[18] A. Varga, "OMNeT++ Simul. Man., v5.0," OpenSim, Tech. Rep., 2016.

[19] ——, "The OMNeT++ Discrete Event Simulation System," in *Proc. 15th European Sim. MC*, 2001.

[20] J. Wang, J. Beu, R. Bheda *et al.*, "Manifold: A Parallel Simulation Framework for Multicore Systems," in *Proc. 2014 IEEE ISPASS*, 2014.