Retrofitting Memoization: An Exploratory Study

Joël Ledelay
University of Twente
PO Box 217, 7500 AE Enschede
the Netherlands
j.ledelay@student.utwente.nl

ABSTRACT

ActFact is a software development company which delivers Enterprise Resource Planning (ERP) as part of a Software as a Service (SaaS) product. As part of their effort to improve the scalability of their software stack, they wanted to investigate the possibilities for them to use memoization techniques to reduce unnecessary repetition of expensive function calls, such as database access and parsing and evaluation of expressions. This project aims to investigate different memoization techniques, in order to determine which of them best suits the company's needs. Their comparative performance have been measured between the methods, as well as the implications that implementing each of these techniques would have on the rest of the system. Based on these metrics, a recommendation was made to employ method level memoization as outlined in this paper.

Keywords

Memoization, caching, optimization, Java, scalability

1. INTRODUCTION

1.1 System overview

ActFact [8] is a software development company which delivers Enterprise Resource Planning (ERP) software as part of a Software as a Service (SaaS) product. Their ERP software is also named ActFact. It consists of a core system, written in Java, with a number of extension modules. These modules are customized for each customer. One customer could require a specific financial management module, while another customer could instead want to run a web shop which is connected to the ERP software seamlessly.

New functionality can be added directly in the ActFact user interface (UI), by defining the structure (column names, types etc) for database tables and windows which display and alter the data in these tables.

This makes it simpler for non-experts to develop new modules, since less knowledge about databases (SQL queries, for instance) is required to create a functional extension to the system.

The server side of the ActFact UI implementation can, like most web applications, be thought of as a request pipeline. A request is received, processed and a response is created and shipped off.

Operations involving input and output (I/O), specifically database access, and those performing parsing and evaluation of expressions take a significant part of the time needed to process a typical request, and are thus a major contributor to the experienced UI performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
34th Twente Student Conference on IT, Jan. 29th, 2021, Enschede, The Netherlands. Copyright 2021, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

In some cases, these operations are unnecessarily repeated while processing a particular request, in some rare cases up to hundreds or even thousands of times. These cases were discovered through code analysis.

When performance issues like these are corrected, it is not unusual for them to reappear as a result of a seemingly small, unrelated change to the UI code. Avoiding or mitigating these issues typically involves duplicated state, leading to a more complex architecture and code base, and more defects. These defects were reported by users, and the root cause was determined to be these ad hoc performance corrections.

1.2 Terminology and research goals

In this research, there are a number of terms which are used extensively. They are explained here.

A *cache*, in the context of this research, consists of a key-value map. This map contains method calls (method name and list of parameters) and maps them to method call results (the return value of the method). Retrieving data from this cache is faster than recomputing the result in the case of expensive function calls. Therefore, using the cache can increase system performance.

Memoization is a software engineering technique, in which results of method calls are stored in a cache. Before running a memoized method, this cache is checked to see if the method has already been called with these arguments. If this is the case, the stored result is returned instead. If the result is not in the cache, the method is run, and the result is stored in the cache.

ActFact aims to improve scalability and reduce the code complexity of the ActFact software stack. These performance issues must be corrected if scalability is to be improved. Reducing code complexity and reducing the amount of defects requires replacement of these ad hoc solutions by a more robust solution which is less complex for the developer.

In order to achieve this, ActFact would like to investigate the possibility to add a memoization implementation on a request scope level.

The possibility of adding a memoization implementation on a request scope level is investigated through analysis of three possible ways of implementing memoization, namely an ad hoc approach, a generalised class level approach and a generalised method level approach.

2. RESEARCH QUESTIONS

The possibilities for adding a memoization implementation are investigated on the basis of the following research question.

RQ1*: How can unnecessary repetition of expensive function calls, such as database access or expression parsing, be avoided?*

This research question can subsequently be split into multiple sub-questions:

RQ1.1 *Where does unnecessary repetition of expensive function calls occur within the ActFact system?*

RQ1.2 *How to ensure that correctness is not compromised?*

RQ1.2.1 *How can it be proven that the expression parsing will return the same result with similar (but not equal) inputs?*

RQ1.3 *How to ensure that minimal code changes are needed in the existing code?*

These research questions were chosen to help identify areas which could benefit from memoization and different methods of implementing memoization. They also help identify performance metrics, such as execution time or their implications for the rest of the system.

3. RELATED WORK

Memoization is a technique which has had significant research done into it in recent years. For instance, J. Mertz et al. [1] recently published a paper in which they provide an overview of the benefits and challenges of applying memoization. They identified three main tasks which are required to apply memoization, which have been used as a guideline for performing this research. These steps are choosing which computations to cache, implementing caching decisions as caching logic and defining a consistency strategy.

S. Wimmer, S. Hu et al. [7] developed a tool which automatically memoizes pure, recursive functions – that is, functions which have no side-effects and provides proof of equivalence. Since this is only applicable to pure, recursive functions, this tool cannot be applied here, since the functions investigated are not free of side effects.

In recent years, research into approximate memoization techniques has been performed [5]. These techniques sacrifice precise result for performance gains. While promising for many applications in which there may not be a single best result, such as a search result or displaying trending posts on social media, this is unfortunately not applicable to this project, since correctness is paramount here. This is the case because ActFact runs administrative software, in which returning an approximately correct result is simply not an option.

P. Prabhu et al. [2] developed a tool which is employed in the context of search algorithms. In search algorithms, in many cases, different solution paths are looped over. If memoization is applied, parallelized access to the memoization data structure is often the main performance bottleneck. This is solved through development of a tool which provides access to a partial view of the contents of this data structure. This way, parallelized access to this data structure is made possible, increasing performance. While this might be an interesting future step for optimization of the memoization algorithms investigated in this paper, implementing such an optimization is out of scope for this project.

Memoization is widely used across multiple languages. For instance, there is built-in support for memoization in Prolog, in the form of tabling [9]. Other functional languages, such as Haskell [10], offer functionality to memoize functions as well.

In Java, memoization can be done in multiple ways. For instance, one could apply ad hoc memoization, by memoizing each method separately [6]. Alternatively, the entire class may be memoized, but this could create significant overhead for classes which contain both methods with high execution times, which are called relatively infrequently, as well as methods with low execution times which are called often. Instead, a generic memoization technique can be applied [11], though this requires changes at every location the methods in this class are called.

While libraries exist for memoization in object-oriented languages [12], it is quite difficult to identify methods which can benefit from memoization, as these methods are often complex. Furthermore, analysing whether any side effects occur in these methods can be time-consuming and error-prone. Research has been done into tools which serve to aid developers in identifying which methods may benefit from memoization [3]. Identification of such methods is, however, only the first step in implementing memoization.

In an effort to aid the developer in detecting methods which may benefit from memoization and providing insight as to how to implement memoization for these methods, a tool was developed [4], which performs dynamic analysis and finds methods which might be overlooked by other analysis tools. This tool was not used, since the company already gave pointers towards which methods could benefit from memoization. These pointers were sufficient to investigate the possibilities of retrofitting memoization in the company's software architecture.

4. METHODOLOGY

A number of steps were identified through which this research is conducted. They are detailed in this section.

4.1 Researching the ActFact code base

4.1.1 Investigating repetition of expensive methods

It must be clear where unnecessary repetition of expensive function calls occur within ActFact. This is needed for two reasons. If it is not known where unnecessary repetition of expensive function calls occur, the implications of developing a solution cannot be investigated. Furthermore, a suitable solution cannot be developed if it's unclear where in the system it should be used.

As a means to find out where this repetition occurs, code analysis is performed on specific parts of the system, which were identified by the company as potential performance bottlenecks.

4.1.2 Proving correctness with similar inputs

Since expression parsing contributes significantly to the experienced performance, this section focuses on those methods which perform this expression parsing, namely *evaluateLogic(Evaluatee source, String logic)*.

In many methods, not all fields and methods of all arguments are used. Because of this, differences in the arguments might not always cause a change in output. This means that if some fields of some arguments are changed, the output could be unchanged. If this is harnessed effectively, the stored cache keys don't need to store all fields of the key Object, but only the ones which affect the output of the method. By doing this, the keys in the cache can cover multiple different possible inputs, increasing the amount of covered inputs with the same cache size, while decreasing the amount of space taken for a cache of the same size.

In ActFact, some of the user session state is stored in a class named Ctx. This class contains a list of contextual values, such as identifiers for which user is logged in, which window is currently opened and many more such values. This list does not have a set size and there are many optional values. Furthermore, these values may change relatively often.

Many classes within ActFact contain a field with the Ctx type. Among those classes is the *Evaluatee* class. Code analysis will be performed on the *evaluateLogic* method in which expressions are parsed, with the goal of identifying whether cache keys for this method can be optimised.

4.2 Researching different methods

In this section, the basic requirements for implementing memoization are discussed, for each of the memoization techniques investigated. The memoization techniques investigated are ad hoc memoization, class level memoization and method level memoization, as shown in Figure 1. These techniques are further illustrated and explained in their respective sections.

In short, ad hoc memoization alters the calling code, inserting caching logic in the method(s) which are to be memoized.

Class level memoization creates a dynamic proxy, which works similarly to the target instance, with one difference: all method calls on that class are intercepted, and caching logic is added. For the caller, it is as though they are calling the methods on the original object, but all methods in the class are memoized.

Method level memoization is similar to class level memoization. It also creates a dynamic proxy which intercepts all method calls. However, it only applies caching logic to specific whitelisted methods, whereas the other methods are called as they would normally.

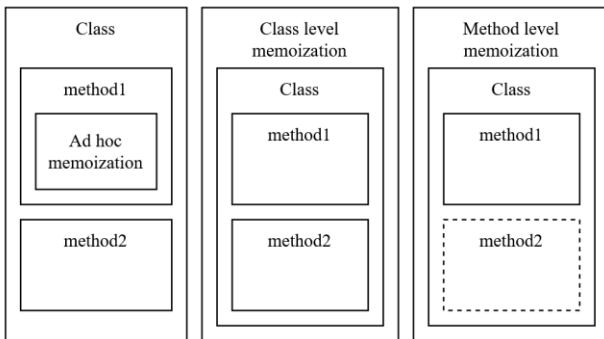

Figure 1. Memoization techniques

4.2.1 Requirements

In a key-value cache, cache consistency is an important topic to consider. Consider the following example, illustrated in Figure 2.

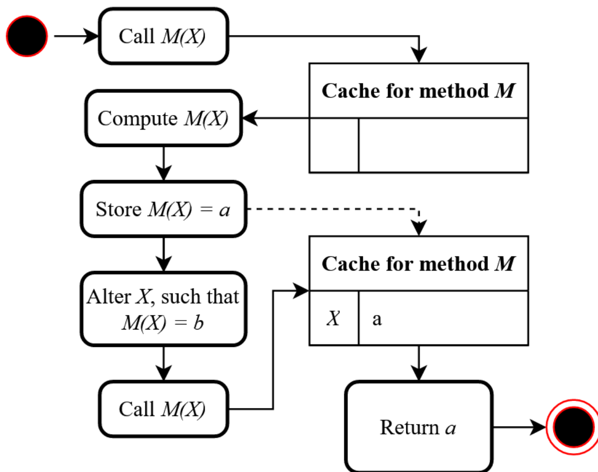

Figure 2. Cache inconsistency. Expected result is *b*.

Method *M*, an expensive method, is being memoized. On one of the runs of *M*, it receives *X* as an argument. *M(X)* is run, and the result, *a*, is stored in the cache. Now, a change is made to *X*, such as changing a field of *X*. Now *M(X)* would return *b* if computed again. If *M(X)* is called now, *a* will be retrieved from the cache, though the correct answer would be *b*.

There are a few solutions to this problem. One option is to invalidate cache entries after a certain amount of time has passed. This is not an option, since cases can still happen where incorrect results are returned, which is not acceptable.

Another option is to invalidate cache entries when the stored arguments are altered. This requires monitoring all the locations where the stored Object may be altered. This is a viable option for classes in which the only way to alter a field is through setters. However, this requires the corresponding getters to return an immutable copy of the field, which is not the case in ActFact's code base.

A third option is to store immutable copies of the arguments which are to be cached. This requires creating a (deep) copy of the argument, which in itself is an expensive operation and is not supported by all classes. Furthermore, this requires the arguments' *equals* method to be suitable for memoization – in particular, for a function *f(x)*, if two arguments x_1 and x_2 are compared, if $f(x_1) \neq f(x_2)$, then $x_1.equals(x_2)$ should return false. An example of a case in ActFact where this is not satisfied is given. Because of this, this option is not suitable.

In the case of *evaluateLogic*, one of the arguments, *Evaluatee*, has a subclass, *PO*, which has an *equals* method which is not suitable for memoization. This equals method does not compare the *Ctx* field, so if two nearly identical *PO* objects with differing *Ctx* field are compared, they are considered equal, but the *evaluateLogic* method might return different results. Since this *equals* method cannot be replaced, another solution is needed.

The option that will be investigated is storing immutable (partial) copies of the key object. These copies should contain all relevant fields and implement an *equals* method which compares these relevant fields. An implementation for this is proposed.

4.2.2 Ad hoc memoization of n-argument methods

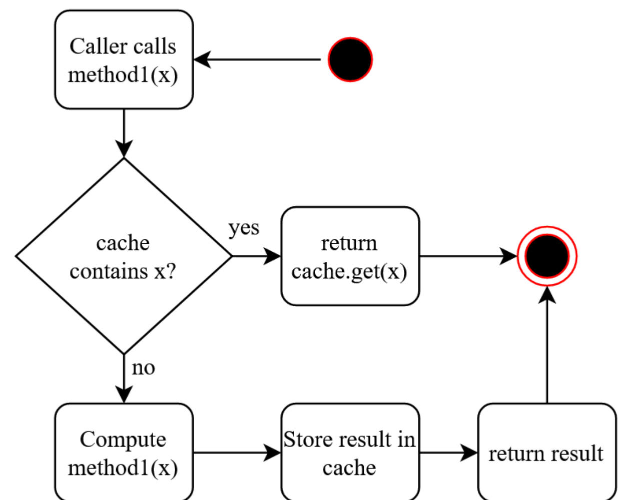

Figure 3. Ad hoc memoization

Ad hoc memoization of n-argument methods is the first technique to memoize method calls. The implication of this is that the calling code must be altered. Rather than computing and returning the value, the cache is first checked to see

whether the input argument(s) are already in cache, as shown in Figure 3. This means that on subsequent runs with the same input argument, the result does not have to be recomputed, but can be retrieved from the cache instead.

The implications of implementing memoization in this manner is that the calling code is altered, leading to more complex and difficult to maintain code.

4.2.3 Generic class level memoization

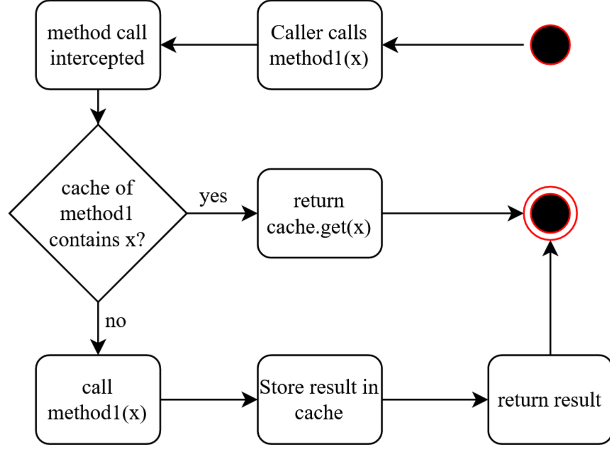

Figure 4. Class level memoization

Memoization is implemented on the class level by defining an interface which contains all methods of the class. At the place where the class is initialised, the class should be initialised through a dynamic proxy instead.

For the calling code, it looks as though they are calling methods on the target interface, but under the hood, the proxy intercepts all method calls on this interface, allowing addition of functionality such as memoization, as shown in Figure 4.

By doing this, no other changes in calling code are required, and all caching code is stored in a centralized location.

A major downside to this technique is that all methods inside the target class are memoized, including the methods with side effects, such as setters.

4.2.4 Generic method level memoization

Since the generic method level memoization is heavily based on the class level memoization technique, it is very similar in behaviour, aside from one main difference. Rather than memoizing every method in the target class, only specific methods are memoized, which are inserted into a list of 'whitelisted' methods. If one of these methods is encountered, the cache is consulted, similar to the class level memoization. Otherwise, the method is called as normal. An illustration of this is shown in Figure 5.

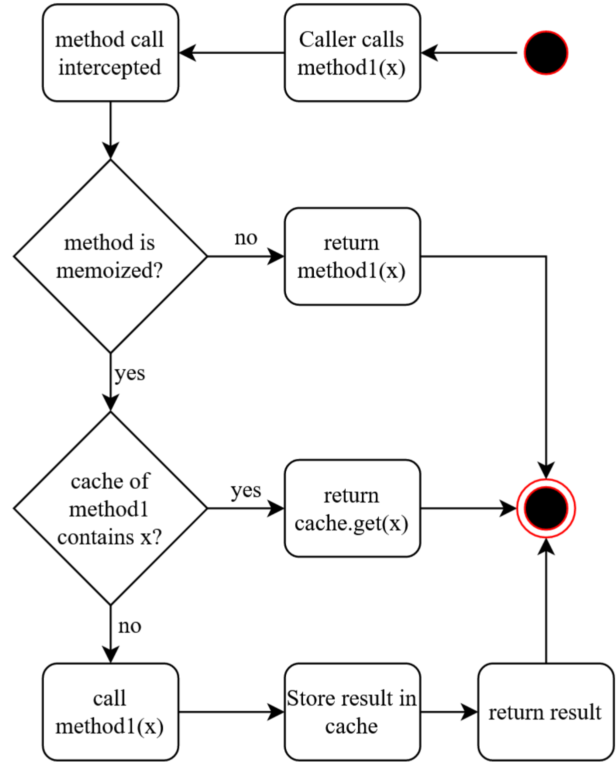

Figure 5. Method level memoization

4.3 Testing environment

These memoization implementations are compared to one other by testing them in a testing environment.

The testing environment consists of a simple scenario with two methods. One method, *expensiveMethod(int a)*, has a runtime of around 100 ms. This method is a model of methods within ActFact which have long execution times, such as I/O operations or methods which perform parsing and evaluation of expressions. The second method, *cheapMethod(int a)*, has a short runtime, performing only a simple arithmetic operation. This method is meant to model methods which are encountered in ActFact which have short execution times.

In the test cases, a mix of methods will be called with long/short execution times, with varying input parameters. In the context of ActFact, it is expected that the majority of method calls have a short execution time, with relatively few expensive method calls. Because of this, the amount of expensive method calls are much smaller than the amount of cheap method calls in the tests.

Though these methods will not perfectly model the situation in ActFact, it is expected that they will serve to indicate comparative performance between the memoization techniques, and as such are a useful tool to analyse which memoization technique will best suit ActFact's needs.

Testing directly on the ActFact code base is avoided here, because otherwise, the requirements outlined in section 4.2 would have to be implemented in order to perform these tests.

A JUnit [13] test class was written which tested the comparative performance of the three memoization techniques.

5. RESULTS

5.1 Researching the ActFact code base

5.1.1 Investigation repetition of expensive methods

Identification of locations where unnecessary repetition occurs was done by contacting the supervisor from the company. They pointed towards a situation which is very common in ActFact, in which unnecessary repetition (dozens of times) of one method, *evaluateLogic*, was observed. Code analysis showed that this method could benefit from memoization. Furthermore, another method, *parseLogic*, was identified which could also benefit from memoization, potentially increasing the performance further. This analysis was done by inspecting the code of *evaluateLogic* and replicating the repetition by running through one of these scenarios.

5.1.2 Proving correctness with similar inputs

While analysing the code for the *evaluateLogic* method, it was discovered that the *Ctx* values contained within the *Evaluatee* do affect the output of the method, but only if these specific values were requested in the expression which is to be parsed.

This means that the cache key can be abstracted to only contain specific *Ctx* values, decreasing the size of the cache entry and allowing multiple different *Evaluatees* with very similar *Ctx* fields to be treated as equal, if the differing *Ctx* values do not appear in the expression. However, distinguishing between *Ctx* values which are used and those that are not requires parsing the expression String. This is considered a relatively expensive operation.

5.2 Researching different methods

5.2.1 Requirements

The idea behind the solution proposed in this section is to have an immutable copy of the source object. The immutable copy should then be used as the cache's key. It should be possible to compare keys of the same type to each other.

An interface *Memoizable* was proposed, in which an abstract class *MemoizedKey* is contained. This interface should be implemented by the classes which are passed as arguments to memoizable methods. The interface should contain a single *toKey* method, which converts the source instance to an immutable instance with the *MemoizedKey* type.

The abstract class *MemoizedKey* should contain a method to compare keys. One of the main challenges here is constructing the *MemoizedKey* objects in such a way that comparison between two objects is possible in a generic manner.

This is yet to be implemented in a generic, reusable manner. This is one of the challenges that has to be overcome in order to retrofit memoization into ActFact's software architecture.

In this research, it is assumed that all input is immutable, and can therefore be copied directly into the cache. The company wishes to use memoization on a request scope level. Therefore, it is unlikely that the arguments passed as parameters will be altered during a request.

The called method and its input parameters are combined into a *List<Object>* which is used as the cache's key. Since the Objects contained in this list are assumed to be immutable, the problem of cache consistency is nullified.

5.2.2 Ad hoc memoization

Implementation of the ad hoc memoization technique was done as such. First of all, a cache needs to be created, which maps input arguments to output values.

```
private Map<List<Object>, Object> cache = new
HashMap<>();
```

Then, the method which is to be memoized is altered, such that this cache is used.

```
private ReturnType expensiveMethod(int a, boolean b,
SomeClass c) {
  ArrayList<Object> args = new ArrayList<>();
  Collections.addAll(args, a, b, c);
  return (ReturnType) cache.computeIfAbsent(args, list
-> {
    // previous method content
    // ...
    return value;
  });
}
```

Inside the *computeIfAbsent* block, the input parameters passed along to the method are also available, so this wrapping code is the only thing needed to implement memoization in this case.

For memoization to be implemented for a second method with similar input types, a second cache should be added to the class. In principle, for methods with different input parameter types, the same cache *could* be re-used, although this exacerbates the problem of maintainability of the code.

5.2.3 Class level memoization

Class level implementation is implemented using a dynamic proxy. On a conceptual level, the proxy 'pretends' to be the target interface. When a method is called on this instance, the proxy intercepts the method call and is able to alter the functionality of this call, for instance by adding logging or time measurement code, or in the case of this project, memoization. This is illustrated in Figure 4.

Instantiation of these proxy objects is simple. Rather than instantiating the target class directly, it is wrapped in a *memoize* call and cast to an interface which contains all methods which are to be called on this object.

```
SampleInterface foo = (SampleInterface)

  ClassMemoizer.memoize(new SampleClass(1, "foo"));
```

Afterwards, this *foo* Object can be used similarly to an instance of *SampleClass*, since *SampleInterface* would have definitions of each of the methods in *SampleClass*.

The major downside of this is that this would clutter up the code base with interfaces for each class requiring memoization.

ClassMemoizer.memoize returns an Object, which contains all methods available in *SampleClass*, but these methods cannot be invoked directly, since the returned type is Object. In order to invoke any method, the returned Object has to either be cast to an interface (as above), or the corresponding Method has to be found first, as such:

```
Object foo = ClassMemoizer.memoize(new SampleClass(1,
"foo"));
Object value = null;
try {
    Method expensiveMethod =
foo.getClass().getMethod("expensiveMethod", int.class);
    value = expensiveMethod.invoke(foo, 1);
} catch (NoSuchMethodException | IllegalAccessException
| InvocationTargetException e) {
    e.printStackTrace();
}
```

This is clearly not ideal, since anywhere an implemented method is called, a try/catch block is needed. It's much more concise and simple to cast the resulting Object to an interface which contains all methods implemented by SampleClass.

Another downside of this approach is that in this case, also cheap methods or methods with side-effects are cached, again resulting in potentially incorrect results. A solution is proposed

to this problem, which is presented in the following memoization method.

5.2.4 *Method level memoization*

Method level memoization is functionally identical to class level memoization, aside from one key difference. Rather than memoizing all methods within a class, only specific methods are selected to be memoized. When other non-memoized methods are called, the method calls are intercepted, but then the original method is called anyway. For memoized methods, the caching logic is used, as shown in Figure 5.

First, the whitelist is created. In the example implementation, this is done by filtering the methods of the *SampleInterface* on some properties, such as method name and/or parameter/return types. Then, this array of whitelisted methods is passed along to the constructor.

```
Method[] whitelist = Arrays.stream(
SampleInterface.class.getMethods()).filter(m -> {
  String name = m.getName();
  Class<?>[] types = m.getParameterTypes();
  return name.equals("aMethod") ||
    name.equals("bMethod") &&
    types[0].equals(int.class) &&
    types[1].equals(String.class);
  }).toArray(Method[]::new);
SampleInterface foo = (SampleInterface)
MethodMemoizer.memoize(new SampleClass(1, "foo"),
whitelist);
```

Ideally, it would be possible to annotate memoizable methods, such that the developer only needs to add an *@Memoizable* annotation in order to enable memoization of the method.

5.3 Test results

Each test consisted of 1,000 runs of the expensive method, and 1,000,000 runs of the cheap method. There was no significant difference between the runtime of 1,000 expensive runs with no runs of the cheap method and the runtime of 1,000 expensive runs with 1,000,000 runs of the cheap method, the differences being so small that they could not reliably be distinguished from the random variation of runtime between runs, despite there being 1,000 times as many runs of the cheap method.

Performance testing indicated that all three memoization techniques are very similar in performance, aside from a few cases. If the cache size is larger than the amount of distinct inputs, all three techniques are near identical in performance. If the cache size is smaller than the amount of distinct inputs, the ad hoc memoization had superior performance, because no limitation on cache size was implemented there. However, since the size of the ad hoc cache is not restricted and managed, this technique suffers from the drawback that it is less space efficient – it does not take into account memory limits, and its performance will decrease as the cache gets larger, as the time taken to look up a cache record increases as the cache gets larger.

The class level memoization suffers from one major drawback when compared to ad hoc memoization and method level memoization, namely that it results in unexpected behaviour when presented with non-pure methods. One such example would be a method which reads and increments the value of a given field and then returns it. On each call of the method, a different result should be returned. When this method is cached, it results in changed program behaviour.

The performance of these three memoization techniques was tested by running a number of different runs, with differing cache sizes and input range for each run. Here, the input range is defined as "the size of the set from which input is randomly drawn". In practice, this meant that an integer was drawn from a uniformly distributed sequence from 0 (inclusive) to the upper bound (exclusive).

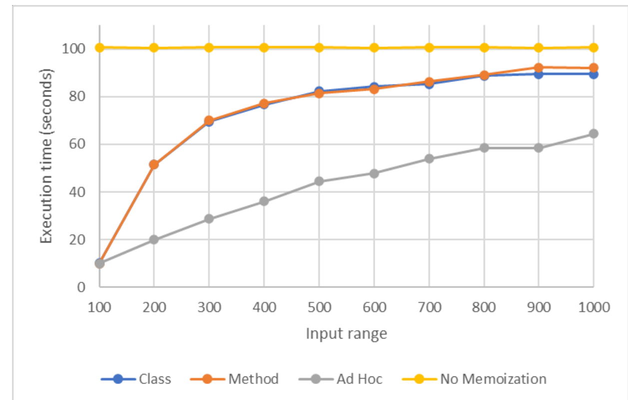

Figure 6: Performance comparison of different memoization techniques at a cache size of 100, with 1000 runs.

Figure 6 shows that the performance of each memoization technique is strongly dependent on how many possible distinct inputs are generated, as expected. Ad hoc memoization vastly outperforms the other memoization techniques when the maximum cache size is vastly different from the amount of distinct inputs, since it does not limit the size of its cache like the other techniques do.

The class level and method level memoization techniques perform similarly. This is as expected, since they use the same technique, with the minor difference that the method level technique adds one if-statement which checks that the called method is in the whitelist.

Figure 7 shows that as the cache size approaches the amount of run tests (1000), the performance of the class level and method level memoization techniques approaches that of the ad hoc memoization. This shows that when configured correctly, class level and method level memoization do not suffer from performance penalties compared to an ad hoc approach, or they gain benefits in the form of space complexity.

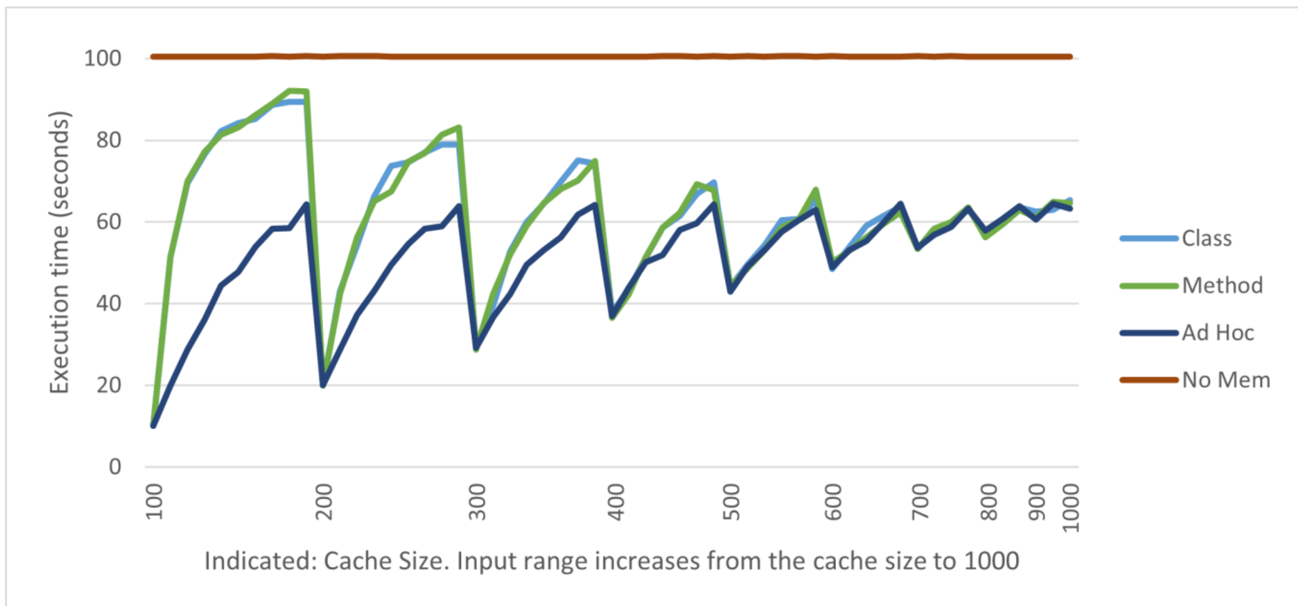

Figure 7: Performance comparison of different memoization techniques for differing cache sizes and input ranges

6. CONCLUSIONS

Based on the results of the requirements identified for each of the methods and the performance and correctness testing, it is clear that the class level memoization technique is not suited for use within ActFact systems. The classes in ActFact don't only contain pure methods, but also contain methods which are intended to have side-effects. By using this technique, these side-effects are eliminated, which results in the program's behaviour changing if this technique were to be implemented.

For the ad hoc memoization technique, a different issue arises, namely maintainability. Since this method requires the developer to change the calling code to incorporate memoization, it increases code complexity, reducing maintainability and readability of the code. While the performance is good, these negatives outweigh the benefits of implementing this method, as compared to the current situation in ActFact, where already several different ad hoc performance solutions exist. The main difference between the current situation and the situation where ad hoc memoization is applied is that there is a single solution, rather than all sorts of separate solutions. The benefits are too small to justify retrofitting this technique into the existing code base.

This leaves the method level memoization technique. This technique suffers from the drawback that it requires interfaces, which contain all the methods usually called on the memoized object.

In the current architecture of the system's application programming interface (API) layer, which is composed of base interfaces, to which developers add functionality by creating classes which implement these interfaces, it would mean that an additional interface should be added for each class, which contains all the methods which developers added. This would require developers to define their methods in multiple places when they add them, which is error-prone and undesirable, since it reduces maintainability of the system. Furthermore, the objects which are to be memoized are currently not constructed in a central location, which means that additional code should be written which manages construction of memoized objects. This code needs to be present at each location where objects which are to be memoized are constructed. This further increases system complexity, reducing maintainability and readability of the code.

However, a new API which is meant to replace the PO layer is currently under development, which addresses among other things the issues which were just described. The new API includes automatically generated interfaces for these developer-added features, which mitigates the need for additional interfaces to be added. Furthermore, the new API also contains an object builder, which constructs instances of the different objects. Since this builder is the only place in the new API where objects are constructed, this is an ideal place to apply the method level memoization. However, in order to do this, the process of constructing a whitelist of methods to memoize must be automated, since these builders are automatically generated by the new API. One possible way to do this is to annotate each memoizable method with an *@Memoizable* annotation, which can be parsed by the API generator and converted into the proper method whitelist for the method level memoization. Another possible method is to define which methods are memoizable through the ActFact UI – since most of the system is defined through database definitions, this can also be extended to include a list of methods which are memoizable.

Because method level memoization requires no changes in the code where business logic is developed, it has minimal impact on the complexity of the code which is commonly worked on by developers.

7. DISCUSSION

7.1 Research questions

While a grounded recommendation is made as to which memoization technique, if any, to apply, some of the research questions have not been fully answered.

The question of where unnecessary repetition of (expensive) method calls occur within the ActFact system was only partially answered. While some possible areas of exploration were identified (database access and expression parsing), due to time constraints, the decision was made to focus on the main goal of this research, rather than performing a broad exploration of which exact methods would benefit from memoization. The goal of this research was never to implement and integrate these techniques into the live system of ActFact within the allotted

time. The areas which were identified already provided a few challenging points, which had to be addressed.

Research question 1.2, regarding correctness, was similarly not answered fully. While correctness is paramount in the administrative software ActFact is running, developing a solution for ensuring correctness proved to be more complex than initially assumed. Addressing research question 1.2.1 similarly proved too complex to model and solve in the proof of concept version. Since the goal of this research was to explore the implications and possibilities of retrofitting memoization, this is no great setback. This research question was aimed towards an optimisation of the basic caching strategy, and was therefore not absolutely required for the purpose of investigating the feasibility of retrofitting memoization into the existing system.

While the analysis described in section 4.1.2 yields interesting results, they are highly specific to the method which is investigated. In a highly general solution, this is not easy to implement in a clean way. Since this project's intent was to explore the possibilities and the feasibility of different solutions, further optimisations to the cache are out of scope for this project. However, these kind of optimisations are an interesting focus for future research on this topic. In particular, dynamic analysis of optimisations like this could yield useful tools for optimising cache performance.

The question of how to ensure minimal code changes are needed in the existing code has been mostly answered. Some additional changes will be required to ensure correctness, as explained in section 4.2.1, though for the most part, the changes needed will most likely be made in the API layer, which is not commonly worked on by developers. This means that the impact for the developers will still be minimal.

7.2 Research validity

In this research, a number of assumptions were made, in order to make the project feasible to implement in the allotted time. These assumptions could affect the validity of this research.

For instance, the assumptions made on runtime of expensive methods are likely to be inaccurate – it was assumed that an expensive method runs for 100ms, which is likely inaccurate. If expensive methods in ActFact run in far less time, this might affect the observed performance of the memoization techniques. A better indication of the observed performance of the current system could have helped in this regard, though it was chosen to not do this due to time constraints. Existing ad hoc solutions should have been identified and removed, if a proper indication of non-optimised performance of these expensive methods was to be obtained. However, this proved to be infeasible due to time constraints.

Other assumptions, such as the amount of runs of expensive vs. cheap methods could similarly be made more accurate through profiling the existing system. As the indicated runtime of these methods are already inaccurate, though, rectifying this does not solve the problem of inaccurate performance measurements without also improving the assumption of runtime of the expensive method. These performance measurements were designed to verify that these techniques did indeed improve performance, as well as give an indication on the effects of cache size and input range on performance. They were never intended to be a fully accurate representation of memoization performance on ActFact's systems.

Aside from assumptions, there was also a slight bias towards the method level memoization. The company which offered this project prefers general solutions over ad hoc solutions in most cases, stating to their developers that they prefer work which is done right rather than work which is done quickly.

There are possible improvements to be made over the simple ad hoc solution presented in this paper, which could have made it more suitable for use within ActFact systems. In an ad hoc solution like this, the managing code could be extracted to a separate class, reducing the impact of this solution on code complexity, and other changes such as bounding the cache size could also have been applied. These additions and changes were not made, partially due to the view of the company towards ad hoc solutions, and partially due to time constraints. Because of this choice, the ad hoc solution was presented as a much inferior solution, although the actual difference between the solutions in terms of suitability might be much smaller. Still, since the company prefers highly generalised solutions over ad hoc solutions, it is expected that the method level memoization technique would still be more suitable.

7.3 Remaining challenges

There are still a number of challenges to be overcome before method level memoization can be applied to ActFact code. These challenges are outlined below.

First of all, correctness must be ensured. An interface similar to the one proposed in section 4.2.1 could be implemented. Alternatively, the (entire) cache could be invalidated when the state has changed sufficiently, since this cache is meant to be used on the request scope level. When a different request is handled, the previous contents of the cache are no longer valid. In order to prove that this preserves correctness, it must first be proven that within a request scope, the objects which are cached are not changed while they are in the cache.

Secondly, the challenge of integration remains. Since a new API is under development, the PO layer of Actfact is gradually going to be replaced by the new API. There are two options here. The first option is to integrate this memoization technique into both the old and the new API. This is, however, not ideal. Implementing this technique into the old API would require changes to a large part of the PO implementation, which increases code complexity. Furthermore, this would require extra investment into something which is to be replaced in the foreseeable future. It seems to be more efficient to integrate deployment of this technique into the deployment of the new API. This has the benefit of not requiring much extra work when compared to making the changes required to implement the new API, while adding additional benefits in terms of performance.

Another challenge related to the smooth integration of this technique is (partial) automation of method selection and code generation. If this method is to be integrated into the new API, it would be ideal if the methods which are to be memoized could be annotated with an *@Memoizable* annotation, which are then automatically added to the whitelist of methods to memoize. An alternative would be to integrate selection of these methods to the existing system, through defining a database table which stores references to the methods which can be memoized.

If these challenges are addressed, the method level memoization technique would be a beneficial addition to the ActFact architecture, which can be added with relative ease.

8. REFERENCES

[1] Jhonny Mertz, Ingrid Nunes, Luca Della Toffola, Marija Selakovic, and Michael Pradel. 2020. Satisfying Increasing Performance Requirements with Caching at the Application Level. *IEEE Softw.* (October 2020), 0–

0. DOI:https://doi.org/10.1109/ms.2020.3033508

[2] Prakash Prabhu, Stephen R. Beard, Sotiris Apostolakis, Ayal Zaks, and David I. August. 2018. MemoDyn: ExploitingWeakly consistent data structures for dynamic parallel memoization. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, Institute of Electrical and Electronics Engineers Inc., New York, NY, USA, 1–12. DOI:https://doi.org/10.1145/3243176.3243193

[3] Hugo Rito and João Cachopo. 2010. Memoization of methods using software transactional memory to track internal state dependencies. *Proc. 8th Int. Conf. Princ. Pract. Program. Java, PPPJ 2010* (2010), 89–98. DOI:https://doi.org/10.1145/1852761.1852775

[4] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance problems you can fix: A dynamic analysis of memoization opportunities. *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. OOPSLA* 25-30-Oct-, (2015), 607–622. DOI:https://doi.org/10.1145/2814270.2814290

[5] Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni. 2018. Temporal Approximate Function Memoization. *IEEE Micro* 38, 4 (July 2018), 60–70. DOI:https://doi.org/10.1109/MM.2018.043191126

[6] Tom White. 2003. Memoization in Java Using Dynamic Proxy Classes - O'Reilly Media. 1. Retrieved November 21, 2020 from https://web.archive.org/web/20150908034250/http://www.onjava.com/pub/a/onjava/2003/08/20/memoization.html?page=1

[7] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. 2018. Verified Memoization and Dynamic Programming. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 579–596. DOI:https://doi.org/10.1007/978-3-319-94821-8_34

[8] ActFact Cloud Business Software | De ERP voor bedrijfsprocessen. Retrieved November 27, 2020 from https://www.actfact.com/

[9] SWI-Prolog -- Manual. Retrieved November 21, 2020 from https://www.swi-prolog.org/pldoc/man?section=tabling

[10] Memoization - HaskellWiki. Retrieved November 21, 2020 from https://wiki.haskell.org/Memoization

[11] Do it in Java 8: Automatic memoization - DZone Performance. Retrieved November 21, 2020 from https://dzone.com/articles/java-8-automatic-memoization

[12] com.google.common.cache (Guava: Google Core Libraries for Java 17.0 API). Retrieved November 21, 2020 from https://guava.dev/releases/17.0/api/docs/com/google/common/cache/package-summary.html

[13] JUnit 5. Retrieved November 28, 2020 from https://junit.org/junit5/