Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Application Level Caching with Automatic Invalidation

Relatore: dr. Matteo Alessandro Dominoni
Co-relatore: dr. Axel Uhl

Relazione della prova finale di:
Raul Bertone
Matricola 707248

Anno Accademico 2013-2014

ABSTRACT

The caching solutions available today for application-generated objects are difficult to use as they offer little help for the management of stored entries: the duty of keeping them up to date is left to application, which is a common cause of errors and makes code maintenance difficult. In this work we investigate how to automate entry invalidation for caches in an object oriented environment. We develop a model for memoization-like caching, which is applicable not just to referentially transparent functions but to any cachable procedure. We apply this model to the case of Java methods by implementing *AspectCache*, a caching framework for POJO which provides automatic invalidation of the entries. *AspectCache* exploits aspect programming techniques and code weaving to cache selected methods and to keep the stored entries up to date. It does so by establishing *dependencies* between an entry and the data accessed during its computation. The framework's simple programming model requires only minimal modifications to the existing code and no specific maintenance of the cached methods during the application development. It works without increasing the algorithmic complexity of the method it operates on and invalidation of an entry is performed in O(1) time; its memory footprint increases linearly with the number of recorded *dependencies*, which is in turn proportional to the input size of the algorithm represented by the method.

Table of Contents

1 Introduction

This thesis addresses the problem of simplifying the management of a cache of Java objects via the automatic invalidation of stored entries.

When an application contains a computational intensive algorithm, a common and straightforward technique to increase performance is to memoize[1] its results to use them later instead of re-executing the algorithm.

The first challenge when using this approach is the identification of stored results, so that, when the same algorithm input values are repeated, the corresponding entry can be found. An algorithm can only be memoized if it is a referentially transparent function (as defined by Søndergaard and Sestoft [23]), and therefore deterministic. For a given input, the output of a deterministic function will always be the same, so a cache entry can be identified by the input values that generated it.

If Java methods were referentially transparent, the problem could quickly be solved creating a key by hashing the method's input arguments, as is the case for functional languages [17]. In fact in languages like Scala, Clojure and Groovy, which like Java run on the Java Virtual Machine but are to various degrees functional, memoization is trivial [9].

However, a Java method is not necessarily a referentially transparent function and therefore not all methods can be easily, if at all, memoized, thus making automation difficult. A method can possess a state (e.g. an internal counter), so that it might not be a function at all; it might not be deterministic (e.g. methods that make use of the system

1 For an in-depth discussion of the differences between caching and memoization, see “Caching Functions” at p. 11. In this work we will generally employ the term caching, reserving the use of memoization for those cases where we want to stress a difference among the two techniques.

clock or of a random number generator); a deterministic, functional method might still produce non-irrelevant side effects and therefore not be referentially transparent. The biggest difficulty derives from yet another difference: during its execution a method can access non immutable data, for example in a database. When any of this data is modified, the algorithm will potentially return different results for the same input: the values stored until that moment in the cache will no longer be valid and will have to be deleted. We will show that such situations can be modelled by considering the method a referentially transparent function, with the caveat that the function itself is not fixed but can be modified and substituted by a different one. Finally, not all the method's formal parameters have the same semantic value of a function input, but can instead represent data.

In consequence of this very relevant differences, automating the memoization process is not straightforward and can only be accomplished to a point. Firstly, it is necessary to identify which methods can be memoized and which of their parameters are to be used in building the key that will be associated with the cached value. Secondly, changes in a datum accessed by the cached method during its execution must trigger the invalidation of all the cached values that used it. While the first challenge concerns only the cached method itself, the second has no precise boundaries, as data can potentially be changed by any part of the application's code, perhaps by one bearing no obvious relation to the cached method itself.

The caching solutions for application-generated objects available today are difficult to use, as they offer little help for the management of stored entries: the duty of keeping them updated is left to the application, which is a common cause of errors and makes code maintenance harder [18]. As such, they do not provide automatic invalidation and do not offer any specific support for memoization, leaving the problems introduced

above to be solved by the application developer. Both generic caching frameworks like Memcached [16] and those specifically developed for Java like Apache's Java Caching System [14] or Oracle's own Java Object Cache [10], as well as many others (EhCache [8], Guava Caches [12], JBoss Cache [15]), are at their core a simple key-value mapping. They present the developer an API to populate the cache and retrieve the entries, but they do not provide tools for automating the construction of a key from a method's arguments. Some differentiate themselves from their competitors by specializing in caching of a specific kind of data, like DB queries, Java Server Pages, HTTP requests or POJO, and focusing on features such as scalability, persistence and replication. None of them offers support for the automatic invalidation of entries, which have to be deleted either manually or by setting standard replacement policies (e.g. FIFO, Least Recently Used, Time to Live, etc.).

The only software that stands out from the others is Spring Cache [22] which, despite the name, is actually a memoization system for Java methods. Its programming model is based on the decoration of methods with Annotations and, while not completely able to automate the key construction, it can simplify the task significantly. Spring Cache too leaves entry invalidation totally in the hands of the application developer.

The first contribution of this thesis is the study of the differences between Java methods and referentially transparent functions, and how the gap can be bridged to allow for memoization. We show that key construction cannot be completely automated because of the partially arbitrary semantic meaning of method parameters.

Also the problem of monitoring data for triggering automatic invalidations cannot be solved in the general case: all data sources would have to be included (DB, file system, input streams, class fields, etc.), but in practice many of them are not under the direct control of the application, and as a consequence monitoring changes there is often not

possible. While support for several external data sources can in theory be achieved on a case by case base, only class fields are always available for monitoring from within the application.

Based on this study we develop an in-memory, in-process memoization framework for Java methods, which we call AspectCache. It automates the construction of keys from the method's arguments and handles their semantic ambiguity by allowing the developer to explicitly indicate it.

Data monitoring, even just for class fields, is a cross-cutting concern, as it can potentially involve the whole application. AspectCache deals with it using aspect oriented programming techniques to locate the points in the code where class fields are accessed, and code weaving to automatically inject the necessary monitoring routines.

The tests we conduct show that the performance of AspectCache is asymptotically optimal. Its memory footprint is linear in the dimension of the number of dependencies between monitored data and cache entries, while the overhead it causes on the cached method execution time grows linearly with the number of data read operations.

We start in the next Chapter by investigating the differences between a Java method and a referentially transparent function. The resulting model represents the foundation on which our memoization framework is based. In Chapter 2 we present the distinguishing features of AspectCache's implementation and in Chapter 3 we test its performance. Finally in Chapter 4 we discuss possible future studies and features to enhance the framework's capabilities. Appendix A contains a short manual for AspectCache.

2 Theoretical Model

Our objective in this section is to define the theoretical model for a cache with automatic invalidation. We will then apply this model to our target implementation context – a bolt-on caching framework for Java objects – and address the issues that arise.

2.1 Caching Functions

In discussing our theoretical model we could start from that of a “plain” cache and just introduce automatic invalidation on top of that. While correct, we believe this approach not to be optimal as it presents the model as a flavour of caching, while its defining feature – the automatic invalidation of entries – brings it much closer to memoization, so that it is more appropriate to consider it a special case of the latter rather than the former.

Caching and memoization are distinct techniques that nevertheless share some characteristics. They are both obviously forms of temporary storage and are based on a key-value pairing. The procedures that are eligible for memoization are all referentially transparent functions (RTF), that is, procedures which can be substituted with their results without altering the program's behaviour. Conversely, a procedure does not need to be a function to be effectively cached. For example consider a web browser cache, which is applied to the procedure that associates a URL to the data therein contained. The URL can, at different times, be associated with different data (for example older and newer versions of an image), so that identical request to the web server will produce

different result. This uncertainty is at the heart of the trade-off that comes implicitly with using a cache, speed (or bandwidth) is gained at the expense of data *freshness*.

However, as variable as the return values of a procedure can be, if we believe it to be suitable for caching it is because we expect those values to be *stable*, that is, to change slowly in relation to the number of times they are used. Moreover, if the values stored in the cache can be safely used instead of performing a new invocation, without compromising the program execution, the cached procedures must also be referentially transparent. This means that, if we suppose the values do not actually vary over time, the cached procedures are indeed RTFs, and can therefore be memoized. We can therefore think a cachable procedure as a *variable* RTF (vRTF). Here lies the only true difference between caching and memoization: the latter has under its control all the input of the RTFs it is applied to, which, therefore, does not appear to change. On the other hand, a cache is aware only of a part of the input, the one that is *explicitly* used in the invocation. When part of the non-monitored (*implicit*) input of a cached procedure changes, the procedure itself appears to be modified. In fact, when invoked with the same explicit input, it potentially returns a different result. In our model we choose to see the implicit part of the input of a cached procedure as an integral part of the procedure itself. A first consequence is that an invocation is completely defined by the explicit input alone, which is therefore enough to build the key that will be used to identify the stored entry.

We believe this model, which sees cachable procedures as vRTF, to be a solid choice because it is not only able to explain why a procedure is cachable, but can also justify the need to introduce the concept of freshness. We can therefore define caching as a particular case of memoization, applied on variable referentially transparent functions.

To this point we are still not able to effectively memoize vRTF, and we had to impose the implicit input to be immutable, which is a very limiting requirement. To lift this

requirement and be able to include all vRTF, in the next section we will introduce the concept of entry invalidation.

2.2 Invalidation

Up until now we required data accessed by the procedures to be immutable, so we could consider it part of the procedure itself. We now relax this constraint and let data vary freely. If we consider data as being part of the procedure, the semantics of a modification in the data is that of transforming the current procedure into a new, different one. The old procedure ceases to exist: repeating an invocation with identical input before and after the update might lead to different results. Because of this, some entries stored in the cache are no longer valid and must be *invalidated*.

Not all the cached values returned by a procedure that has elevated an invalidation need to be removed. A procedure will often read several pieces of data during its execution, most likely different sets when invoked with different input arguments. When a cached procedure reads a specific piece of data we say that the resulting cache entry has a *dependence* on it. Only those cache entries that have a dependence on the datum that has changed are now invalid.

To delete all and only the relevant cache entries and to do it in a timely fashion, the framework must monitor all data and react when a change happens. We note how this can be seen as an example of *reactive programming* or, more generally, as an implementation of the *observer pattern*.

When talking about caching, validity and staleness are used somewhat interchangeably. We find this confusing and misleading, so we take a moment to discuss the meaning of the two terms as they are intended in this work. We prefer to talk about staleness in relation to caching and reserve the use of validity for memoization. A cache

entry becomes stale based on some condition *local* to the cache data structure (for example after a certain time), but that does not imply it is no longer valid. In fact, the remote procedure that generated the entry has not been invoked, and its status remains therefore unknown. In the case of memoization, there is no degree of uncertainty as the status of an entry is always a known quantity. This ambiguity is partially justified by the fact that the idea of a modifiable value is associated with caching, because memoized values are usually immutable and do not happen to be invalidated as the functions that produced them do not normally change.

2.3 Data as Implicit Input

The caching model we described is not the only possible one. We will now introduce a partially different approach: comparing it to our original model will allow us to discuss and better motivate our choices. Previously we considered data as being an integral part of the procedure: we will show that this unintuitive and seemingly arbitrary choice was the only possibility.

To insure that a procedure could be seen as a function, we had to assume that all data is immutable. Now we instead choose to consider the same data as an – implicit – part of the procedure's input, so that the immutability requisite can be lifted. Consequently, when we now look at a procedure we see data as part of its input variables.

The semantics of a data modification is, in this new model, different than before: instead of replacing the existing procedure for a new one, it simply represents a different invocation of the existing, unmodified, procedure. This is exactly the same effect that would be caused by a change in the explicit input variables.

While this model is sound, in the sense that it allows to see a given procedure as a

function, it is unsuitable for caching. A cache entry is, in both models, the result of a procedure call. To correctly identify and retrieve it from the cache all the input arguments used for the procedure invocation are needed. Since in the second model data constitutes an implicit part of the input, it too is required for a cache look-up. Specifically, when assembling a key to perform a cache look-up, the information we need are the data locations (and their values) which *would be* accessed if, instead of the cache look-up, the procedure call was to proceed normally. It is intuitively evident that, in the general case, any attempt to retrieve this information without actually executing the procedure would be equivalent to solving the Halting Problem. This renders the use of a cache pointless: if the procedure has to be re-executed at every invocation, the cache would never be used.

This difficulty is due to a fundamental difference between a function and a procedure: the former receives all its input as part of the call, while the latter can obtain part of the input it requires by accessing data that is not provided with the invocation. This second model does not acknowledge such difference, handling both sources of input equally, and therefore it cannot provide a solution for our problem.

2.4 Implementing the Model

Now that we have defined a generic caching model, we can tackle the problem of translating it into a Java implementation. Our objective remains building a framework that allows adding caching to individual, existing procedures. It must be possible to do so with minimal or no modifications to the application code and in particular without having to worry about caching in any phase of the application design. While in the second part of this work we will examine the implementation details, we focus here on explaining how the concepts we developed translate when applied to our object-oriented

context of choice and what problems might arise.

The model we defined has functions as the smallest cachable units: in Java the construct that gets the closest is certainly the *method*, and from this we will set off in our conversion. Once we select a method for caching, we first need to inspect its invocations and correctly identify those that happen to be repeated. Secondly, the method's return value must be associated to its corresponding invocation and all this information stored into a data structure from where it will be retrieved when needed.

A second step concerns the data the cached method accesses within its control-flow, for in our model that data forms an integral part of the cached procedure. We say that a read operation creates a *dependency* of the method's output on the datum. If any of this data is modified, regardless of when that happens, all existing dependencies will be broken and the relevant cache-entries invalidated (that is, all those cache entries during whose calculation the modified fields were read).

The monitoring activities just described create an obstacle to our objective of creating an easy to use, bolt-on framework: the code points that need to be modified easily become too many for the task to be tackled manually and, most importantly, they are often hard to identify in the first place. Let us look at two of the worst cases. When we need to intercept a method's return value we cannot simply take the naïve approach of modifying the explicit return points in the method's body because, if an exception is thrown within the control flow of the method, the intercept code will never be reached and the return value (in this case a *Throwable* object) will be missed. Indeed, from the caching perspective, an exception should be treated as no special case: the *Exception* object is just seen as the method's return value for that particular set of input arguments. It is therefore necessary to insert the monitoring code not in the method itself, but wherever the method is invoked from. As a second example consider the task of finding all class fields that could potentially be read within the cached method's control-flow

(we are always interested in them because they might represent data for our function): this requires completely exploring each possible execution path that has root in the cached method. These examples show that, even for a single method, the necessary changes could be spread all over the application's code in seemingly unrelated places. A modification to the method could affect any and all of them. Also, when writing new classes or adding new software modules, the developer would have to be aware of the fact that, somewhere, the interface she is implementing might be used in the control flow of a cached method and therefore appropriate steps need to be taken (that is the class fields of the new implementation must be monitored too). The software's maintainability would be severely compromised. To overcome this problem the task of *injecting* the monitoring code throughout the application has to be automated. Indeed, the need to monitor such diverse things is a perfect example of cross-cutting concern and this is why we choose to use aspect-programming techniques to make the code injection automatic, namely with the help of AspectJ [2].

A second obstacle is due to the semantic ambiguity of the parameters of a Java method. While in a mathematical function the input can only represent variables, in a Java method it might also contain data. The semantic meaning might appear obvious to a human reader, but it is ultimately arbitrary and implicitly assigned by the developer: there is no objective way to discriminate between data and variables in the method's formal parameters and so no possibility to automatically tell them apart. As a consequence, the framework must provide the developer a way to manually specify the semantic meaning of a cached method's parameters.

The implementation of the model encounters a third problem when monitoring class fields: not necessarily all class fields read within the control-flow of a method are relevant to the method return value. Common examples include logging-related fields and thread synchronization data structures. If dependencies are created on these fields

the caching logic will not be compromised, and the framework would still be able to operate with reduced efficiency. Unnecessary invalidations could be fired, causing the deletion of valid entries from the cache. To improve efficiency we want our framework to allow manual exclusion of class fields from the monitoring routine.

2.5 Related works

In this section we examine research works that present similarities with ours.

TxCache [18] is the work that resembles ours the most. Although its declared objective is to provide transactional consistency to a database-backed cache of application generated objects, it uses automatic cache invalidation as an intermediate step. To our knowledge it is the only other existing framework that provides automatic invalidation in an imperative environment (the current implementation is in C++). Like AspectCache it uses a simple programming model where existing application methods can be selected for augmentation by the framework. TxCache's biggest difference, and most important drawback compared to our solution, is the need for all the data used by the cached functions to be stored in a relational database. This requirement, arguably necessary to ultimately obtain transactional consistency, limits the applicability of the solution. The RDB approach also influences the tracking of dependencies between cache entries and data: the latter live in a database and it is not accessed directly by the application, but through queries. As a consequence, when a database entry is selected through a query, a dependency will be created not just on that specific tuple, but also on all the database tables accessed while elaborating the query. A modification to any of those tables will then trigger an invalidation for the tuple. Indeed, a table modification will invalidate all queries performed on it, even if they would not be affected by the change. Our solution

acquires a finer invalidation granularity, removing from the cache only the entries actually affected by a data modification.

Furthermore, transactional consistency – arguably the most important functional advantage of TxCache over AspectCache – can theoretically be realized in the latter too, as we discuss in Chapter 4. However, without storing the data in a RDB, the potential usefulness of the feature remains to be seen.

DITTO [7] is an automatic incrementalizer for dynamic, side-effect-free data structure invariant checks. When DITTO sees a change in a monitored data structure, it performs a check on it to verify the integrity of some invariant property. To speed up the checks, its algorithms are designed to be incremental, that is, they re-run only those parts of the check that might be influenced by the change. It shares with AspectCache the use of Java bytecode weaving techniques to monitor changes in data structures. Also, the algorithms it uses to check invariants are made incremental through the use of memoization.

Self-adjusting Computation [1] techniques are able, through the use of a modified language compiler (for the ML language) to automatically transform functions into incremental algorithms: when a change to a special data location happens, the consequences for previously calculated results are automatically resolved through so-called change propagation. Only those parts of the algorithm that are actually influenced by the change are recomputed, resulting in a substantial speed up.

Like in the case of TxCache, the main aim of the approach is not automatic cache invalidation, which is only a necessary intermediate step. Nevertheless the SAC model presents several similarities to the one needed for “simple” automatic cache invalidation. Both require to explicitly identify the functions' input – in SAC this is

achieved with a static analysis of the code – and data, which must be written in specific memory locations called modifiable references.

While AspectCache cannot perform the translation of a given function into an incremental algorithm, a similar effect can be attained exploiting its self-recalculation feature (see also p. 35) and manually subdividing the function we aim to incrementalize into smaller sub-functions. When an invalidation is thrown, it will propagate to all and only the relevant sub-functions, which will be re-run. Still, the result is not as fine grained as with SAC.

In addition to the research efforts reported above, some form of automatic cache invalidation is present in a few software.

Spring Cache. Spring Framework [21] is an open source application framework for Java. Its caching abstraction framework [22], much like AspectCache, allows to add memoization to Java methods by decorating them with specific Annotations. Once a method is decorated, the construction of a cache-key from its arguments is highly automated and can be fine tuned through the proprietary Spring expression language (SpEL). The cache implementation leverages Spring's aspect oriented framework which does not employ code weaving techniques but is instead proxy based. As a consequence only public methods can be cached. Interestingly, Spring Cache does not impose the use of a particular storage solution for the actual cache entries, consistently with it being an abstraction layer: different “back-end” storage can be chosen and plugged-in by the application developer. It does not provide any support for the automatic invalidation of entries.

IBM's **WebSphere** [13] application server offers a cache invalidation system based on a

publish/subscribe model. When a cache entry is added, some invalidation IDs can be associated to it. When a request to the server causes invalidation IDs to be generated, all objects that are associated with those invalidation IDs are removed from the cache.

The **ASP.net** [3] web application framework has some support for automatic cache invalidation. It provides the developer a specific CacheDependency class that can be manually associated with a cache entry in the role of the dependant, and a set of objects in the role of dependencies. Possible dependencies include other cache entries, file, database query and user input. When one of the dependencies is modified, invalidation of the cache entry is performed automatically.

2.6 Reactive Programming

It is interesting to note that a system for automatic cache invalidation, regardless of how it is implemented, follows at its core an observer pattern: it monitors the input data and when a change occurs it reacts by appropriately invalidating entries in the cache. As such, it shares the model that is at the base of reactive programming. It is not surprising that reactive computing languages and frameworks offer many key features useful for the implementation of automatic cache invalidation. For example FrTime [6] is a reactive programming extension to the Racket functional programming language. It provides, at the language level, the data monitoring features required for automatic invalidation. When an observed input data is modified, a *signal* is created which propagates the effect of the modification throughout the program. In the Microsoft world, the Rx extensions for the .NET platform [19] provide similar features, as do the RxJava [20] and the somewhat dated SugarCubes [4] libraries for the Java language. However, a reactive programming approach can not, on its own, enable a crucial part of

the automatic invalidation logic: the linking between a cache entry and its dependencies has still to be programmed manually and is not created automatically at run-time by the language/framework as is the case for AspectCache.

3 Implementation

In this chapter we will describe AspectCache's implementation and how it manages to solve the two main challenges: associating cache-entries with method invocations; maintaining the dependencies between cache-entries and the data used to calculate them.

We start with an overview of the software's functional behaviour; then we present the programming model and environment requirements; finally we will examine the implementation details of AspectCache's distinctive features.

3.1 Functional Description

The functional behaviour of AspectCache does not differ, when observed as a black box from the host application's point of view, from a “normal” caching service. Every time a cached method is invoked AspectCache performs a cache look-up to check if an entry corresponding to that particular invocation is present. If the entry is present, it is returned to the invoker, without ever executing the actual method. If it is not, the original invocation is allowed to proceed normally and, when the method will return, its return value will first be stored in the cache and then returned to the invoker.

However it is what happens *during* the cached method execution, or more precisely within its control-flow, that differentiates AspectCache from other object caching frameworks: every time data is read, the event is caught and associated with the cache entry currently under calculation. A dependency relation is therefore established between each datum and the entry. Later, when any of those data happen to be modified,

the event will be caught and, by consulting the set of dependencies associated with the modified data, AspectCache will be able to identify all potentially affected cache entries and remove them from the cache. The removed entries might have had several other dependencies and they will be deleted too. This last task is necessary because in the future new, updated versions of the same entries might be stored in the cache and the dependencies of the invalidated versions must not be included.

To be truly transparent for the host application, AspectCache must accept all return values as valid input, including Exceptions. They are treated like any other entry and stored in the cache. In case a subsequent invocation corresponds to a saved Exception, the latter is returned to the invoker exactly as if it had been thrown by the cached method.

Moreover, the host application's concurrency model must be respected and left unmodified by the framework. To achieve this, AspectCache does not employ its own threads but instead piggy-backs on those used by the host application to invoke the cached method: the same thread performs the cache look-up and, in case an entry is not present, proceed without modifications to the execution of the cached method. In particular, threads traverse AspectCache code without ever being blocked or acquiring locks that might block other threads.

3.2 Programming Model and Environment

The two main functions of AspectCache are the caching of Java objects and the automatic invalidation of entries after modifications of the data they have been computed from: of course, both these objectives could also be obtained by adding them

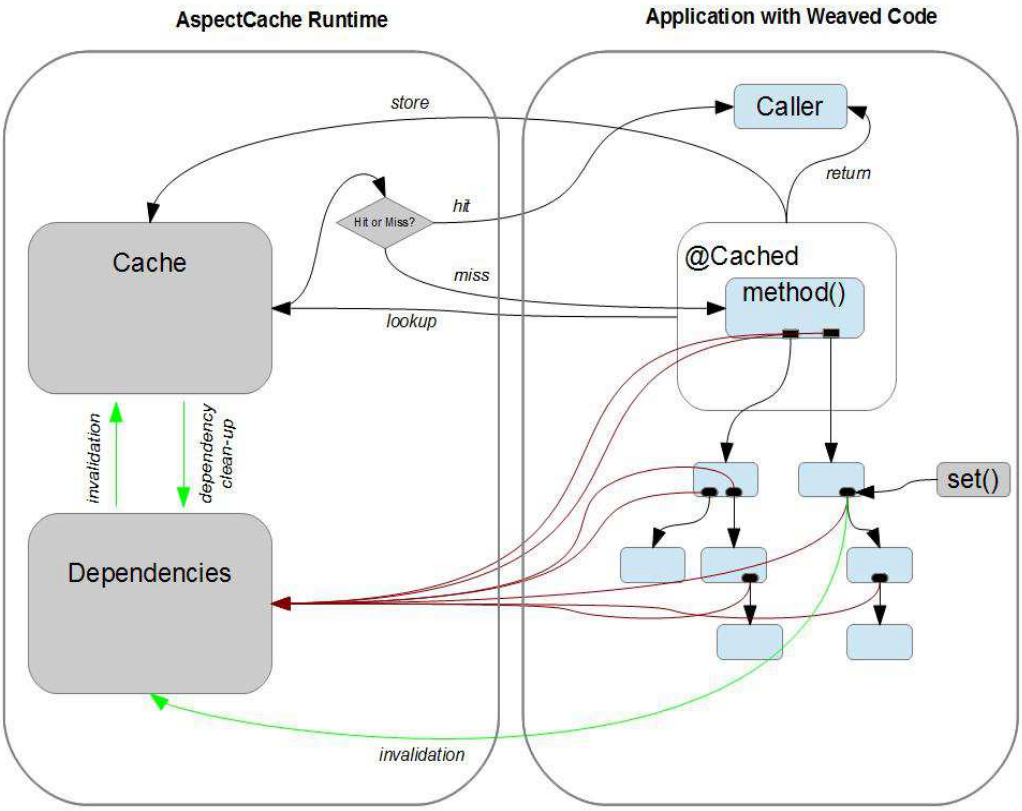

Illustration 1: a) the caller invokes the cached method and AspectCache intercepts the call; b) a cache look-up is performed: if the entry associated with the invocation is present, it is returned to the caller, otherwise the cached method is allowed to proceed as normal; c) within the cached method control-flow, every time a class field or a collection element is accessed, a dependency is registered (red arrows); d) when the cached method completes its execution, first a copy of the return value is stored in the cache, then the value is also passed back to the original caller and the normal execution flow of the application resumes; e) if later one of the data that had been accessed by the cached method is set, an invalidation request is raised (green arrows): the set of dependencies associated with it is retrieved and used to remove the appropriate entries from the cache; f) for each removed entry, the remaining dependencies associated with it are deleted.

manually. A key feature of our framework must therefore be its simple programming model, so that caching with automatic invalidation is added to existing software with minimal effort.

In the simplest use case the only alteration the developer has to perform on the source code is decorating the method she wants to cache with a specific Annotation:

```
01 @Cached
02 public int anExpensiveMethod (int arg1, int arg2) {
03    // method body
04    }
```

AspectCache will then use AspectJ to identify the code-points where modifications are needed and AspectJ's compiler will take care of injecting the required code.

In more complicated use cases some other modifications might be required. While we will describe the details later in this chapter, we anticipate that in all situations the modifications will be minor and will not involve structural alterations of the existing code.

Dependency requirements are kept to a minimum for easy integration, and include only AspectJ. While the framework does not require the use of a specific IDE, and in fact it requires no IDE at all, we point out that AspectJ is very well integrated with Eclipse and as a consequence AspectCache is too, so that few simple steps are sufficient to start integrating AspectCache into a host application (see Appendix A).

3.3 Recognizing Invocations: Cache Keys

The first problem to solve in caching method invocations is to distinguish the

invocations themselves, to create a one-to-one association between them and the cache entries. Starting from an invocation we have to create an ID, or key, by which the entry will then be identified; when an identical invocation will then be repeated it will generate the same cache-key allowing the easy retrieval of a stored entry, if one is still present in the cache. To do so we must first establish when any two distinct invocations are considered equal.

A method invocation in Java is completely and univocally identified by the union of two components: the identity of the method being invoked and the value of the arguments being passed to the method with the invocation. In turn, a method is identified by its fully qualified name [11] and, accounting for the possibility of method overloading, by the number and type of its parameters and by its return type.

The second component, the invocation's arguments, is not as straightforward to handle. In the case of primitive types their values simply correspond to the argument values, while for reference types determining their “value” is more complicated and – as far as caching is concerned – even allows for multiple interpretations as we will discuss later. In the most general case we define the value of a reference type as the values of the referenced object's class-fields and, when these are in turn of reference type, recursively by the value of the objects they reference. Alternatively we can say that the value of a reference type corresponds to the value of its deep-object-graph.

All this values can contribute to the identity of an invocation. However, as discussed in Chapter 1, sometimes not all the method's parameters are intended by the developer as variables and they instead represent data. Consider the following examples.

```
01 List<Person> selectByAge(int minAge, int maxAge, List<Person>
employees)
```

This method selects and returns from the provided list all *Person*s within a certain age range. We invoke the method once and cache the result. Let's imagine that the address of one of the employees included in the *List* is modified. If we now repeat the invocation with the same List instance and the same age range we still expect it to be recognized as identical to the first one and that the cached value be returned. What we do not expect is for a different cache-key to be generated and a second cache entry to be stored. We expect this behaviour because of the method's semantics: only the identity of the list represent a variable, while its content represents data.

```
01 List<Position> findShortestPath(Position start, Position finish,
List<Position> positions)
```

In this second method a list of points (they could be GPS fixes) are ordered so that the total distance travelled to go from start to finish is minimized. Again, we invoke the method once and cache the result. If we then invoke it again with a different List containing a different set of Position instances which all happen to represent the same points as those of the first invocation, we expect the two to be seen as identical. This is because not just the List, but all of its content represents variables (as well as the start and finish Positions), while no data is in this case passed to the method with its arguments.

To a human reader the different semantics of the previous examples should look easy enough to understand at first glance. However, parameter semantics cannot be determined automatically by AspectCache, and must be indicated explicitly by the developer. To facilitate this task the framework provides three different kinds of cache-keys, each one with some strength and weaknesses. It is possible to use only one kind of

key for any single cached method, but different keys can be employed for different methods.

Identity: only the identity of the arguments passed to the cached method are interpreted as variables, while their values are interpreted as data. Invocations are considered equivalent if their arguments are identically-valued primitive types or reference the same object instance. If the reference types passed as arguments for two invocations of the same method reference distinct object instances, they are considered different even if both instances possess the same value. While from a semantic point of view this key cannot fit every situation, on the other hand it does not impose any requirement on the method's arguments (as the other key types do).

Immutable-Arguments: this key type assumes that the arguments passed to the cached method are immutable. The immutability requirement limits its possible uses but also permits all values to be safely interpreted as variables. All arguments' values, primitive types as well as reference ones with their complete deep-object-graph, contribute to generate the cache-key. No argument is interpreted as data. For the sake of performance, this key does not use reflection to explore the object graph, so it is limited by the implementation of the equals() method in the classes involved. It has the smallest construction cost of all the cache-keys and it permits the use of all the optional features which AspectCache offers (they will be described later in this chapter): for these reasons it is the key of choice every time its use is possible.

Serialized: all arguments are considered variables but, unlike the immutable-arguments kind, they are not assumed immutable. They cannot be used directly to build a cache-key, because some of the arguments might get modified, in which case we would end up

with a different key associated with the same entry. Their value at the time the invocation takes places must therefore be saved: this is obtained through the standard Java serialization process that returns a String which in turn is used as a key. This kind of cache-keys trades the requirement of having immutable arguments with that of having serializable ones and can therefore fit distinct use cases.

The choice of a particular cache-key applies to all the parameters of a cached method. It would be desirable to be able to assign individual key types to each parameter as to match their semantic meaning but this is unfortunately not possible at the moment. AspectCache uses Annotations to select the cache-key type for each cached method and Java, up until version 7, does not allow Annotations for the parameters in method declarations (for a discussion of the new possibilities opened by Java SE 8, see Chapter 5).

3.4 Monitoring State

To automatically invalidate a cache entry the framework must know which data had been used during its calculation, that is, which class-fields had been read in the control-flow of the cached method. The class fields are instrumented by AspectJ at compile time and when one is read – and this happens in the control-flow of a cached method – a dependency is registered between the field and the cache entry that is being generated. When afterwards the value of one such class-field is modified, the change is observed by the framework and an invalidation event is created.

However, not all the fields that are accessed by a method are data in the sense we defined in Chapter 1. There are four reasons why a field might not represent data, which define four categories of fields. A field can belong to more then one category. Firstly,

fields that are declared *final* are not a possible source of invalidations because they cannot be modified. This first group is simply ignored by our framework.

Secondly, we consider fields whose values have no consequence on the return value of the cached method. This is a vast category that includes, among others, fields that are used for synchronization purposes. Consider for example a simple semaphore, implemented using a boolean field: threads accessing the semaphore are stopped or allowed to proceed depending on its value. All cache entries created by threads that used the semaphore are invalidated every time the boolean value is flipped.

Thirdly, we find fields that are set within the control-flow of the same cached method invocation. Such fields might represent side-effects of the method, like logging, output or again synchronization. They are also a potential bug as they might constitute an internal state of the method, which would not be a function and therefore would be unsuitable for caching.

Lastly, there are fields that belong to objects which are part of a serialized cache-key. While the key itself is serialized and will not change, the objects it was obtained from can potentially be modified. The latter, by definition, represent variables and not data for the cached method so that, when they change value, they should not provoke any invalidation.

Accepting invalidations by any of these fields is a conceptual error. However, with the exception of those fields that constitute an internal state for the method, if they are included in the monitoring routine, AspectCache can still function properly: the only effect is to reduce the efficiency of the framework which is forced to remove entries from the cache as a consequence of unnecessary invalidations raised by non-data fields.

To reduce the effects of this problem AspectCache allows to manually exclude individual fields by decorating their declarations with a specific Annotation (@NotData), like in the following example:

```
01 @NotData
02 private CountDownLatch cdl = new CountDownLatch(21);
```

Fields thus decorated will not be instrumented and they will not register dependencies or raise invalidations.

Since AspectJ performs its code injection at compile time, AspectCache is limited to work with those fields that are available to the compiler. To find which fields to monitor AspectJ follows all code paths that could be executed within the control-flow of the cached method. This might lead to the Java API or third party code. While AspectJ is able to work on any standard Java source code or bytecode artefacts, there is a number of reasons the developer might want to prevent it from doing so: if only the bytecode is available, and detailed information on the original compilation settings is unavailable, the recompilation performed by AspectJ might produce a significantly different bytecode; the rights to modify third party software could be missing; without the source code it is impossible to manually exclude fields from instrumentation, which might decrease efficiency; all modified third party software, including the Java Runtime Environment, would have to be re-distributed with the application, making deployment more complicated.

Fortunately, this problem is easily solved, for AspectJ can be instructed to ignore specific Packages and in fact, by default, it does not inject code into classes belonging to the Java API.

Moreover, AspectJ is not able to track read or write accesses to array fields and this means AspectCache cannot monitor them. If any arrays are read within the control-flow

of a cached method and they contain data, the framework will not be able to function properly[2]. The problem can be bypassed using Collections instead of arrays.

3.5 Collections

Since Collections are part of the Java API, monitoring them by simply instrumenting them presents the problems we discussed above; also, the standard implementations are internally based on arrays so that they actually cannot be monitored in the first place. AspectCache must then provide a different and specific support for Collections, and it does so with the use of wrappers for the standard interfaces of the Collection Framework.

The wrappers intercept invocations to the Collections and, examining its arguments, are able to understand on which element or elements of the collection a dependency is being established.

Consider for example a wrapped object implementing the List interface on which the following invocations are made:

```
01 lst.get(42);
02 lst.indexOf(obj);
```

When the first invocation is intercepted the wrapper registers a dependency on the element at index position 42. For the second invocation the dependency is registered for the index occupied by the element 'obj', if it is present in the List. If the element is not present no index can be associated with the dependency which is instead associated with

2 There is no theoretical obstacle preventing the tracking of array access. Chen and Chien have proposed an implementation of such a feature [5].

a special register of negative requests.

When the list is in any way modified the wrapper works out which dependencies are broken as a result and issues invalidations accordingly, like in the following example:

```
01 add(5, newElement);
```

As a consequence of the addition of newElement all the elements at indexes ≥5 will have their indexes increased by 1, so all dependencies associated with these indexes will be broken. Also, since a new element has been added, all previous negative indexOf() requests could yield a different result now and their dependencies are therefore broken too. The wrapper then selects all cache-entries affected by the change, sends an invalidation request for them and removes all the broken dependencies.

The structure of dependencies generated by Collections is profoundly different from that of class-field dependencies, and also varies significantly from interface to interface: Collection dependencies are not stored centrally like class-field ones but locally in the wrapper itself.

Collection wrappers are used out of necessity because direct instrumentation is impossible, but it turns out they offer occasion for increased performance: they "understand" the semantics of method invocation and as a consequence can minimize the number of dependencies created. In the previous example, following the indexOf() invocation, several if not all elements in the list would need to be visited and compared, until eventually the looked-for element was found. If each element in the list – and each class-field they possessed – was individually instrumented, many more dependencies would be created instead of just one.

The management of *Exceptions* fired by wrapped *Collection*s requires special care. As the wrapper cannot know in which state the *Collection* was at the moment the normal execution flow was interrupted, it cannot determine which dependencies to create and which to break. After intercepting an *Exception* and before re-throwing it, the wrapper is forced to invalidate all cache-entries that depend on the wrapped collection.

3.6 Other Features

3.6.1 Recalculator

AspectCache can be configured so that cache-entries produced by selected cached methods automatically recalculate themselves when invalidated. This can be useful to better exploit the available resources, performing entries recalculations in off-peak moments. The effect is particularly relevant if invocations are likely to arrive in bursts. If several threads perform the same invocation while there is no corresponding cache entry, each one will start a parallel calculation, wasting resources[3].

When such an entry receives an invalidation, instead of being removed from the cache, it recalculates itself by creating and sending a *Task* to an *Executor.* This *Executor* uses threads from a *ThreadPool* that is independent from the host application. This constitutes an exception to the normal case where cached methods are only executed by the host application's own threads. It can only be used with methods that do not have special requirements for the threads executing them. Recalculator's threads are set to the minimum possible priority available in the current thread group to avoid competing for resources with host application's threads.

3 Another possible scenario, depending on configuration, has one thread performing the calculation and all the others waiting for it to finish. See also *Concurrency Optimization* at p. 37.

3.6.2 Nested Calls

When, as part of the execution of a cached method, another invocation to a cached method is made, whether a recursive call to the same method or a nested call to a different one, dependency management becomes more complicated. We will refer to the two cached methods involved and their associated entries as parent and child. There are two possible cases: the child call might or might not have an associated entry already stored in the cache. If it does not, the child method is executed, so that there are now two cache entries under calculation, and the dependencies generated concern both and should be registered accordingly; if, on the other hand, the child call is already associated to an entry, the child method is not executed and the existing entry is returned to the parent which will therefore not acquire the necessary dependencies. If one of the child's dependencies was now to fire an invalidation, the parent would not be affected. AspectCache solves this problem as follows.

When a nested call is performed a special dependency is created for the parent entry *on its child entry;* then, *w*hile the program is within the control-flow of the child method, dependencies are registered only for the child entry. This way the total number of dependencies is halved and memory footprint reduced. When the child entry is invalidated, before removing it from the cache, AspectCache recursively invalidates all entries that depend on it.

Another advantage of this approach is that, if an entry of the self-recalculating kind gets invalidated, it will first recalculate itself and then, only if the new value is different from the old one, propagate the invalidation to its parent entries.

The union of nested calls and self-recalculating entries allows to transform a method into a *reactive incremental* one. The method must be divided in several parts (which will necessarily still be referentially transparent) and each new method must be individually

cached. When an entry generated by any of those methods is invalidated, it is re-computed and the effects automatically and recursively propagated, so that only the sections that are actually affected by the modification are re-executed instead of the whole original procedure.

3.6.3 Concurrency Optimization

Normally, when a thread requests a cache entry that is not present but for which a calculation by another thread is already under way, it does not wait for the result of the other thread but simply starts another parallel calculation. This behaviour is guaranteed to respect the concurrency model of the host application. If the concurrency requirements allow it, AspectCache can instead stop a thread while there is already an ongoing calculation, forcing it to wait and use the other thread's result, saving resources that would be lost in identical simultaneous calculations.

3.6.4 Threads and Executors

In our discussion so far we always assumed that all the execution of a cached method happened in a single thread. In practice many different, more complex situations can arise. We will divide them in two categories, which we will refer to as *parallel* and *synchronous*.

We call parallel the case in which a thread, while in the control-flow of a cached method, creates one or more children threads – directly or else, for example assigning a task to an Executor – which contribute to the calculation of the entry. In this situation the parent thread must wait for the children to finish, because it makes use of their results and in doing so it logically acquires a dependency on all the data they read. They all, parent and children threads, collaborate to produce the same cache entry and we say they operate within the same cache-context.

In general a child thread is not required to participate in the production of an entry, their semantic function depending completely on the application design, and it is therefore impossible for AspectCache to infer if the cache-context includes a specific child thread or not. Any class whose instances are intended to be executed by a thread must implement the *Runnable* interface. To manually force the framework to extend the cache-context to a specific *Runnable*, the class must also implement a specific marker[4] interface, *MonitoredTask*.

In the asynchronous case a thread executing within the control-flow of a cached method uses the results of computations executed by a different thread that was created *outside* the control-flow: the parent thread had no chance to extend the cache-context to the thread it is using the result of, because it is not one of its children, but still it must somehow inherit its dependencies. While of course this situation can present itself in a variety of ways, Java provides specific support for asynchronous computations through the use of the *RunnableFuture* interface, and AspectCache supports it. If the result of a *RunnableFuture* is to be used from within the control-flow of a cached method, it must implement, like *Runnables* in the parallel case, the *MonitoredTask* marker interface. When the *RunnableFuture* is executed AspectCache will implicitly cache its run() method, registering its dependencies and saving its result. Every thread which will call its get() method will acquire a dependency on the associated cache entry, similarly to what happens with nested calls.

AspectCache does not support other ways of propagating the cache-context besides the two described above.

4 An interface that defines no methods or fields is a marker interface.

4 Performance Testing

A cache's main purpose is to increase the performance of software, allowing to save computational or bandwidth resources in exchange for a bigger memory footprint. However it is not possible in general to quantify this increment because it depends primarily on the usage pattern rather than on the specific software and cache: a cache look-up is usually completed in constant time and therefore the only remaining variable is the hit/miss ratio, that is, how many times on average an entry is read before being removed from the cache.

AspectCache performs a look-up in O(1) time, regardless of the host application or the cached methods characteristics, so the performance advantage is similar to that of a normal cache, as is the difficulty in assessing it.

Two unique elements must be taken into account when investigating the performance of this particular caching framework: firstly, both the registration of dependencies and the execution of invalidations have a computational cost; secondly, on top of the memory needed to store entries we have to add the amount needed for dependencies. Again, the real impact – negative in this case – that this factors have on performance depends on the usage pattern, in particular by the number of dependencies that are on average registered for each entry and, as a consequence, how many of them must be removed after an invalidation. We investigated the asymptotic complexity of dependency registration and invalidation, and the memory footprint of dependency storage.

For this task we developed an ad hoc host application which allows to set precisely and at run-time the relevant characteristics of one of its methods – the one cached by the

framework for this tests. The parameters that can be modified include the number of dependencies created as a result of an invocation, how many entries depend on a class field or collection element and the number of entries actually created. Crucially it is also possible to change the usage pattern, setting the invocation and invalidation frequency.

Measuring execution time in Java is a less than exact process, especially for intervals as short as the ones in question here. At first we tried using *ThreadMXBean.getThreadCpuTime()*, but the cost of this method turned out to be too high, higher in fact than the operations we were trying to measure. The method we employed instead, *System.nanoTime(),* has, according to the Java Specification, nanosecond precision but not necessarily nanosecond resolution, which means there is no upper limit on accuracy. Moreover *System.nanoTime()* returns the elapsed time, not the CPU-time for the current thread, which lowers its reliability. To get partially around this problems we proceeded in each case to average the results over several runs. Furthermore, to reduce the effects of the initial loading process and just-in-time compilation, we always allowed the program to run long enough to reach a “steady” state before performing our measurements.

AspectCache exposes performance metrics at runtime by means of an MBean and the results that follow have been obtained in this way.

4.1 Dependencies Registration

Every time a field is read during the control-flow of a cached method, a dependency has to be registered, adding an overhead time to the method calculation. We measured a method's execution times with and without AspectCache, varying the number of fields read during each run to verify that the overhead grows, as expected, linearly with the

number of registered dependencies. The method itself was designed so that it too has a calculation time which grows linearly with the input size, so that its characteristics do not overshadow what we are trying to measure.

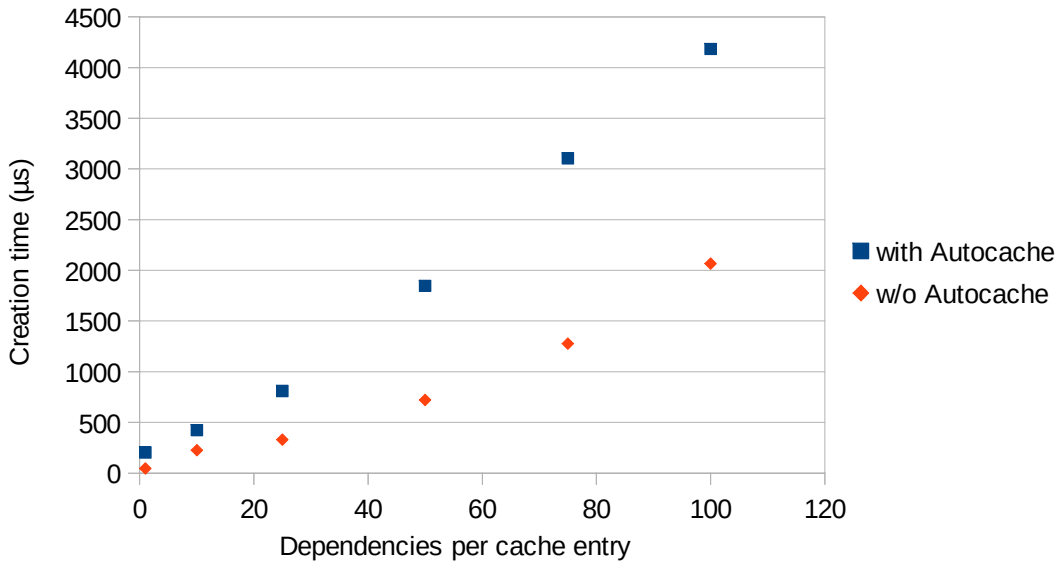

Illustration 2: creation time of 10000 cache entries with varying number of registered dependencies per entry.

4.2 Invalidations (dependencies clean-up)

When a field is modified and an invalidation generated, all cache entries that depend from it have to be removed from the cache and all dependencies that they possessed deleted. If E is the number of cache entries that depend from a field and F is the average number of dependencies an entry has, the invalidation time should be directly

proportional to both parameters and therefore to E*F.

In the first test the host application was set so that from a field depended always one and only one entry. In this situation every invalidation concerned exactly one entry and therefore a known (and fixed) total number of dependencies. In the second test we kept the number of dependencies per cache entry constant, varying instead the number of entries depending from a single field. In both cases the results show, as expected, a linear relationship between the total number of dependencies to be removed and the time necessary to complete an invalidation.

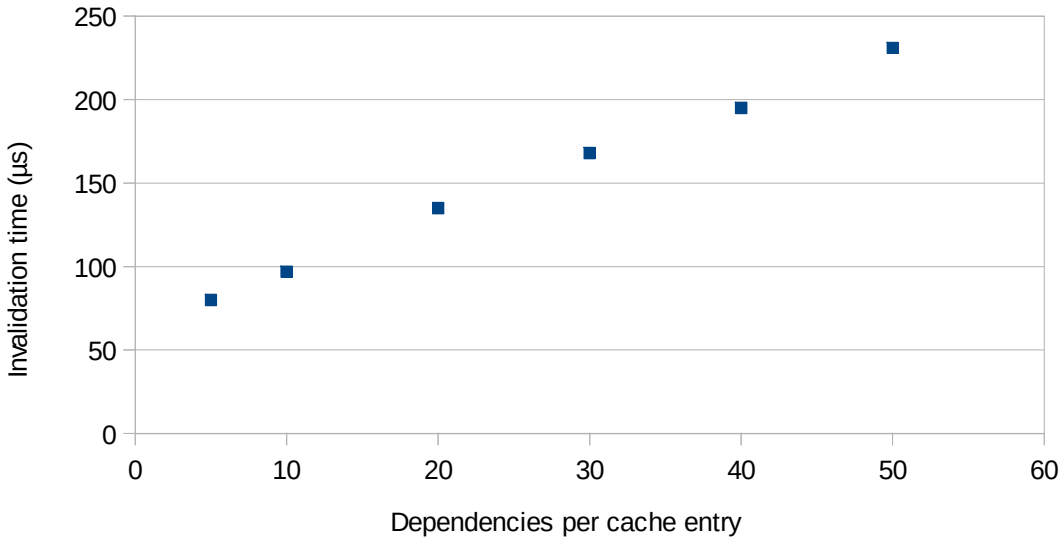

Illustration 3: time required to perform an invalidation for varying values of dependencies per cache entry

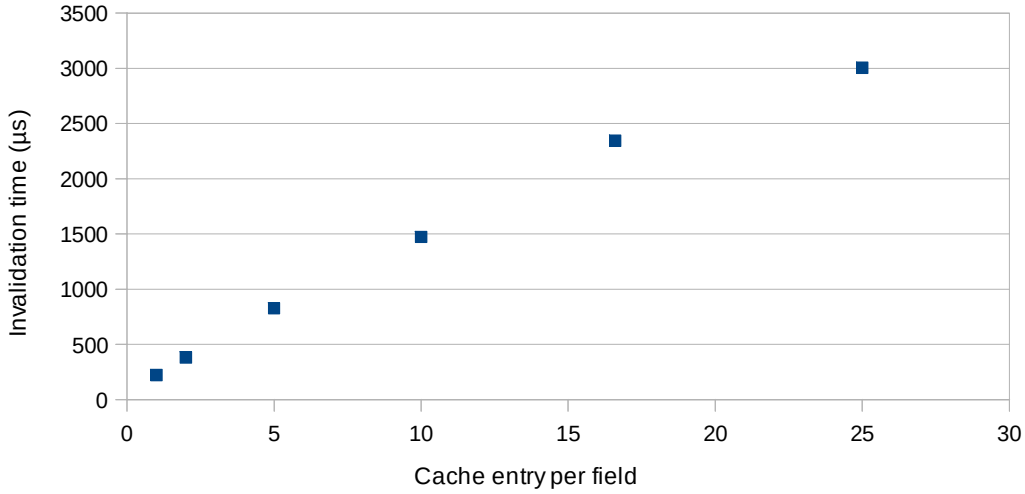

Illustration 4: time required to perform an invalidation for varying numbers of entry per field.

4.3 Memory Footprint

The memory footprint of AspectCache is made up, in its variable part, by the stored cache entries and their dependencies: while the entries footprint depends largely on the host application, we can say more about the contribution of dependencies.

Dependencies in AspectCache come in two flavours, field dependencies and collection dependencies: we investigated their approximate size and whether the linear relationship between their number and footprint held.

It is important to stress that a dependency is an abstract concept that, when implemented, does not necessarily translate into a single entity, be it an object or an object reference, so that its “size” may vary not just from one kind of dependency to the other but also within the same kind. Furthermore, the amount of memory occupied by a

Java object or reference is not specified by the language but depends from the Virtual Machine implementation and the machine architecture[5]. In consideration of these variables the calculated size of a single dependency must be considered a rough approximation.

In every test run a fixed number of entries (10000) was created, varying for every run the number of dependencies that each of them possessed.

Two series of test have been performed, one for each kind of dependency.

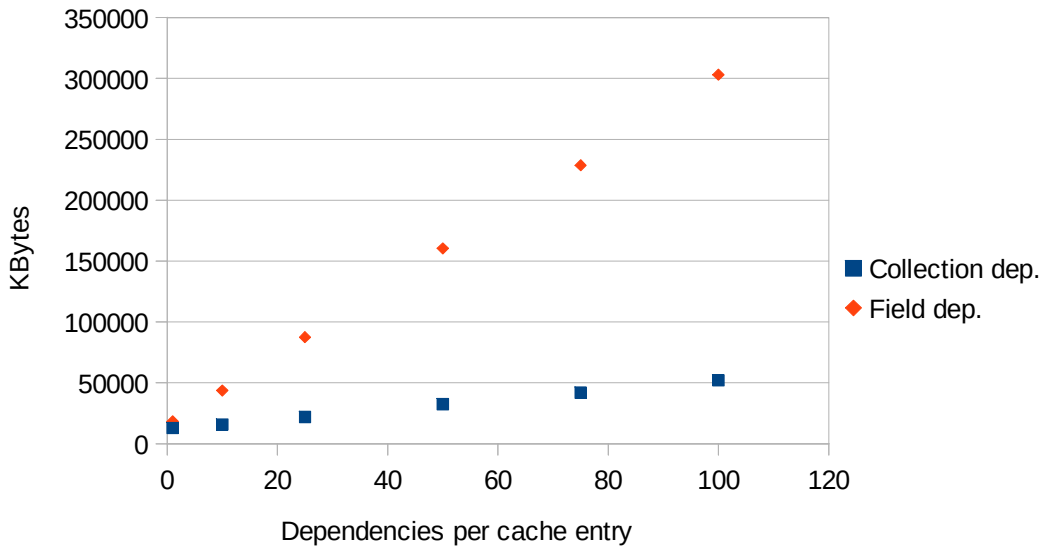

Illustration 5: memory occupied by the dependencies of 10000 cache entries with varying numbers of dependencies per entry.

Results show that the footprint scales linearly with the number of dependencies. The average size of a field dependency is approximately 240 bytes, while that of a collection dependency around 40 bytes.

5 The test were performed on a 64-bit machine running a 64-bit operating system; the Java Virtual Machine used was HotSpot Version 7 Update 51.

5 Future Studies and Development

The implementation of AspectCache is at the prototype stage and therefore several core features are incomplete or absent, so completing their development is the first necessary step. This framework is also a relatively innovative software and it was not certain when we set off whether the Java language and the existing code weaving tools would allow to achieve our objectives, and to do so while keeping the software convenient and easy to use: arrived at this point in development we expect to encounter no insurmountable obstacles along the way. Most of the work that remains concerns collections support, a vast but rather straightforward task, as it requires to write wrappers for all the standard interfaces in the Java Collection Framework.

Furthermore, before AspectCache can be used on any method, without requiring substantial refactoring of its code, it will be necessary for AspectJ to properly support the detection of access to array objects. More specifically, AspectJ is able, in the current version (v. 1.8.0), to detect access to a class field of type array, but not to the single elements of the array.

```
01 public class ArrayExample () {
02    static int[] anArray = new int[10];
03    public static void main(String[] args) {
04          anArray[0] = 42;
05    }
06 }
```

In line number 2 a set operation on the class field is correctly detected, but in line 4 only a get operation on the field is intercepted, while we cannot obtain the set operation on the array element itself and we do not know which index has been modified.

Since AspectJ is an open source project, we do not have to wait for a future version to implement the feature we require, but we can proceed ourselves to do it as part of AspectCache's development, so the missing feature is not an insormountable obstacle.

The release of Java 8, which happened while this work was already in its final phase, presents several new possibilities and challenges. The ability to operate on collections through Streams requires to expand – and possibly in part rethink – the current approach of tracing dependencies for collection elements using wrappers. We expect the use of lambda functions to filter elements will make understanding which elements have been accessed and/or modified in a given call problematic. On the other hand this new way of accessing collections standardizes elements selection through the use of aggregate operations that employ internal iteration. The following two blocks of code select the same elements using internal and external iteration respectively:

```
01 List<Person> roster = new List<Person>;
02 for (Person p: employees) {
03    if (p.getGender() == Person.Gender.MALE)
04          roster.add(p);
05 }
```

```
01 List<Person> roster = employees
02    .stream()
03    .filter(p -> p.getGender() == Person.Gender.MALE)
04    .collect(Collectors.toList());
```

Internal iteration gives us the opportunity to improve the precision of invalidations, and therefore of the whole caching routine, for the case in which a new element is added to a collection. At the moment this event must trigger the invalidation of several entries that accessed the collection but are not necessarily affected by the new element. Suppose, after the roster List is created in the previous example, we modify the source collection:

```
01 employees.add(new Person(Alice, Johnson, Person.Gender.FEMALE));
```

In this case the new element introduced in the collection does not result in a different roster List, but with external iteration we would have no easy way of determining it. When using Streams instead, a strategy that might work in many such cases consists in saving the “query” – represented in the example by the filter operation – and then, when a new element is added, re-run it: if it selects the new object, then the associated cache entry must be invalidated. This approach works only if the query is based exclusively on properties that are local to a single element:

```
01 List<Person> roster = employees
02    .stream()
03    .filter(p -> p.getGender() == Person.Sex.MALE &&
                                    followsJohn(p))
04    .collect(Collectors.toList());
05
06 private boolean follows = false;
07 public boolean followsJohn(Person p) {
08    boolean value = follows;
09    if(p.getName == “John”) follows = true;
10    return value;
```

This example selects male persons (a local property) if they follow a person whose name is “John” (order based property). The introduction of a new element will alter the query results depending on the position where it is inserted, even if it is not itself selected by the filter.

Java 8 also introduces Type Annotations, extending the possibility of applying an annotation to any type *use* instead of *declarations* only [24]. This could make AspectCache significantly easier to use by improving its flexibility. The current cache-keys treat all the formal parameters of a method in the same way, assigning the semantic

value of 'variable' either to their identities or to their values. However, this is a simplification because in reality different parameters can of course have different semantic values. Type Annotations allow to decorate each parameter individually, so that keys could be tailored for every method. Exploiting this new use of annotations would also require AspectJ to be updated to support it, but again, as for the problem of array access detection, this is not outside of our control.

When we discussed TxCache (see p. 18), we pointed out how one of its defining features was transactional consistency. We believe that AspectCache could be adapted to provide some form of transactional consistency and we think that further study in that direction would be worthwhile.
In the case of TxCache, which requires application data to be stored in a relational database, the implementation of transactional consistency is a duplication of a property that is normally guaranteed by the underlying data layer, which would be lost with the use of a cache. On the other hand, in our case, transactional consistency would likely be an additional feature introduced by the framework itself.

TxCache offers both serializable isolation and the weaker snapshot isolation. For an application that does not employ a relational database, and therefore does not have a single point of entry to its data layer, we expect serializable isolation to be impractical if introduced mechanically: the framework would have to superimpose its own thread concurrency model over that of the host application, most likely producing anomalies and generally rendering concurrency control impossible. When using a RDB instead, the two models – that of the host application and that of the framework – come into contact at one point only, making coordination easier.

Snapshot isolation does not impose requirements on thread concurrency and should be quite straightforward to implement through the use of multi-version concurrency

control, for example by storing for each class field multiple versions of its value.

Dependencies and their registration, storage and removal have a severe impact on performance, but they are necessary to automate cache entry invalidation. To limit the overhead due to dependency management as much as possible, we described (see p. 31) how the framework allows to manually exclude class fields that do not represent data and whose values have therefore no impact on the method's return value; the same can be done for collections by not implementing the specific monitoring interface. However, finding the fields and collections to exclude can be tedious and error-prone. We think it could be possible, and indeed very convenient, to automate this process through static data-flow analysis.

Also, it is not possible at the moment to specify for each individual field and collection which cached methods should consider them data, and therefore generate dependencies on them, and which should not. Such an option would bring performance benefits at the cost of making code maintenance more difficult, so we decided to leave it out for now. However, an automated selection process would have no such problems and, being already able to tell if a given method depends on a certain datum, could proceed to exclude it selectively.

6 Appendix A

6.1 Using AspectCache in Eclipse

AspectCache does not require the use of any IDE but, as AspectJ is highly integrated with Eclipse, it is convenient to use this IDE with AspectCache. We describe here the necessary steps to integrate AspectCache in an existing application using Eclipse:

1. *in Eclipse install the AspectJ Development Tool plug-in;*
2. *import AspectCache in the Eclipse Workspace as a Java Project;*
3. *now AJDT must be told which classes have to be weaved with the aspects contained in AspectCache. Do do that, add them to the AspectCache project* Properties → AspectJ Build → Inpath. *Alternatively, if the destination (the project to be augmented by AspectCache) is itself an AspectJ project, you can add AspectCache to its* Properties → AspectJ Build → Aspect Path. *For more information about this step please refer the AJDT guide.*

6.2 Caching a Method

After completing the environment configuration it is possible to start augmenting the existing application with AspectCache. The first step is to select which methods are to be cached. To activate caching for a method, decorate it with the `@Cached` annotation.

```
01 Public MyClass {
02    @Cached
03    public int expensiveMethod (int i, int j) {
04          // method body
05          return value;
06    }
07 }
```

Since annotations on methods are not inherited, the method in the parent class and in the subclasses which override it must be individually decorated.

It is important to note that, since all keys employ the fully qualified name of the type declaring the method, overriding methods are seen as distinct from the overridden ones. We subtype the class created in the previous example and decorate it again:

```
01 Public AnotherClass extends MyClass{
02    @Cached
03    @Override
04    public int expensiveMethod(double d) {
05          // method body
06          return value;
07    }
08 }
```

Now, two identical invocations, one to the overridden method and the other to the overriding one, do not generate the same key and therefore do not reference the same cache entry.

AspectCache allows to tailor the caching behaviour on a per-method base through the use of a set of optional parameters. The options are specified as elements of the `@Cached` annotations. For example:

```
01 @Cached (waitForFresh = true,
02      automaticRecalculation = true,
03      keyType = KeyStrategy.ARGS_IDENTITY)
```

The elements' default values need not stay the same across all the overriding versions, therefore you can enforce different caching behaviors for the same method by calling it from subtypes of the original version.

6.2.1 keyType

This option allows to select the key type used to identify the invocations to the cached method. The possible values are:

```
KeyStrategy.IMMUTABLE_ARGS
KeyStrategy.MUTABLE_ARGS
KeyStrategy.ARGS_IDENTITY
```

For a description of each key type and of their differences, see p. 26.
The default value if the option is not specified is "`ARGS_IDENTITY`".

6.2.2 automaticRecalculation

Boolean option. If set to true, cache entries generated by this method will automatically recalculate themselves when invalidated. The default value is "false".
For a detailed description of the Recalculator, see p. 35.

6.2.3 waitForFresh

Boolean option. When set to true, threads performing a cache look-up which find that another thread is already calculating the cache entry they need, will wait for the latter to finish and use the its result. The default value is "false".
For a detailed description of this feature, see Concurrency Optimization at p. 37.

6.3 Excluding Class Fields

Some class fields might not contain data relevant to the cached method's output. To prevent AspectCache from monitoring them, decorate them with the `@NotData` annotation.

Access (get/set) to fields decorated with this annotation will not be monitored even if happening within the control flow of a cached_method.

As annotations on fields are not inherited, `@NotData` must be applied to the parent class' field and to all subclasses which eventually hide the field.

```
01 Public MyClass {
02    @NotData
03    private String irrelevantField;
04
05    // rest of class body
06 }
```

6.4 Monitoring Collections

Collections are not normally monitored by AspectCache. To force monitoring, the collection must implement the `MonitoredCollection` interface.

This is a marker interface, that is, it contains no methods or fields. Since interfaces are inherited by subtypes, all subtypes of a collection implementing the `MonitoredCollection` interface will be monitored too.

```
01 Public MyCollection extends List implements MonitoredCollection {
02    // class body
03 }
```

6.5 Monitoring Children Threads

By default, only the thread executing a cached method produces dependencies for the cache entry under calculation. That is, only class fields and collections accessed within the control flow of a cached method produce dependencies. To extend the dependency production to other threads, the Runnable or Callable they run must implement the `MonitoredTask` interface.

This is a marker interface, that is, it contains no methods or fields. Since interfaces are inherited by subtypes, all subtypes of a Runnables and Callables implementing the `MonitoredTask` interface will too produce dependencies.

```
01 Public MyTask implements Runnable, MonitoredTask {
02    // class body
03 }
```

6.6 Monitoring AspectCache at Runtime

During runtime, AspectCache exposes some useful metrics through an MBean. The MBean registers itself with the standard platform MBean server and can then be accessed as usual with a JMX Connector (e.g. Jconsole).

Bibliography

[1] Acar, Umut A. “Self-Adjusting Computation”, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005

[2] AspectJ, http://eclipse.org/aspectj/

[3] “ASP.NET Caching” [online], Microsoft Developer Network (MSDN) http://msdn.microsoft.com/en-us/library/ms178597(v=vs.100).aspx

[4] Boussinot, F.; Susini, J. “The sugarCubes tool box: a reactive Java framework”, Software—Practice & Experience, Volume 28 Issue 14, Dec. 1998, pp 1531-1550

[5] Chen, K.; Chien, C.H. "Extending the Field Access Pointcuts of AspectJ to Arrays," 2006 International Workshop on Software Engineering, Databases, and Knowledge Discovery

[6] Cooper, G. “FrTime: a Language for Reactive Programs” [online], http://docs.racket-lang.org/frtime/

[7] DITTO. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 310-319

[8] EhCache, http://ehcache.org/

[9] Ford, Neal. 2014, “Java.next: Memoization and functional sinergy” [online], IBM developerWorks, http://www.ibm.com/developerworks/library/j-jn12/

[10] Franci, A. et al. “Java Object Cache”, Oracle Application Server Containers for J2EE [online], http://docs.oracle.com/cd/B14099_19/web.1012/b14012/objcache.htm

[11] Gosling, J.; Joy, B.; Steele, G.; Bracha, G.; Buckley, A. “Java Language Specification, SE 7 Edition”, Chapter 6 [online], July 2011, http://docs.oracle.com/ javase/specs/jls/se7/html/jls-6.html

[12] Guava, Caches [online], https://code.google.com/p/guava-libraries/wiki/ CachesExplained

[13] Hines, Bill. 2004, “Static and dynamic caching in WebSphere Application Server V5” [online], IBM WebSphere Developer Technical Journal, http://www.ibm.com/developerworks/websphere/techjournal/0405_hines/0405_hines.html

[14] Java Caching System, http://commons.apache.org/proper/commons-jcs/

[15] Jboss Cache, http://jbosscache.jboss.org/

[16] Memcached, http://memcached.org/

[17] Norvig, Peter. "Techniques for Automatic Memoization with Applications to Context-Free Parsing," Computational Linguistics, Vol. 17 No. 1, pp. 91–98, March 1991

[18] Ports, D. R. K. “Application-Level Caching with Transactional Consistency”, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2012

[19] Rx Reactive Extensions, https://rx.codeplex.com/

[20] RxJava, https://github.com/Netflix/RxJava/wiki

[21] Spring Framework, http://projects.spring.io/spring-framework/

[22] “Spring Framework Reference Documentation”, Chapter 29 [online], http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/cache.html

[23] Søndergaard, Harald; Sestoft, Peter (1990). "Referential transparency, definiteness and unfoldability", Acta Informatica 27 (6): 505–517

[24] “Annotations on Java Types”, JSR-000308 [online], https://www.jcp.org/en/jsr/detail?id=308