# 15745 Project Final Report

## Inserting Caches using Compiler Passes to Optimize RPCs between Microservices

**Bhakti Chaudhari <bchaudha>**
**Adarsh Sreedhar <adarshsr>**

## 1. Motivation

### a. *The Problem*

Microservice Architecture as we know, is a widely used software architecture paradigm wherein an application is broken down into a set of loosely coupled execution units, running specific logic and altogether representing a coherent functioning unit. The advantage of using this approach is that each microservice can be scaled independently as demand requires, and promotes fault tolerance, since a failure of one service does not stop the entire application. For example, let us consider a Train Ticketing Application: the services to handle booking and reservation will see a spike in usage on the weekend and drops in usage on weekdays, as people are busier during the same time. These specific services would also need to be scaled then, and then reduced when demand drops. Such an architecture also promotes abstraction, in that a service only exposes certain Application Programming Interfaces while the implementation can be hidden and changed as required.

While the advantages of such an architecture are many, there are some tradeoffs that we need to make as well, when compared to the monolithic architecture. For example, as many services can be functioning in different machines across different regions, this implies that there would be a lot of network traffic between services communicating to each other as well. Continuing our example of the train ticket application, opening the application itself would at-least call these services – login and authentication, ticket history and cancellation and account details and settings. It is interesting to also note the dependencies between the services – only after the login/authentication service is complete, we move on to reading the account settings, and proceeding to fetching history and past bookings.

Such dependencies between services imply that there is regular communication between microservices, which mostly make use of the network for message passing. A poorly designed service architecture will then have the danger of being blocked by network latencies, specifically when there are a lot of messages being passed and the network is blocked. In such cases, it is essential to reduce the communication, either by caching data or by using event-driven architectures. In our Train ticket example, we could cache data such as available trains between cities (after the user contacts the service for the first time). Assuming that the user would like to review the trains again, we could then just send the cached data.

It is not just communication between services, but also data that needs to be written/read to/from databases are significant sources of latencies, and so would benefit heavily from caching such calls, as long as consistency is maintained and faulty data is not stored.

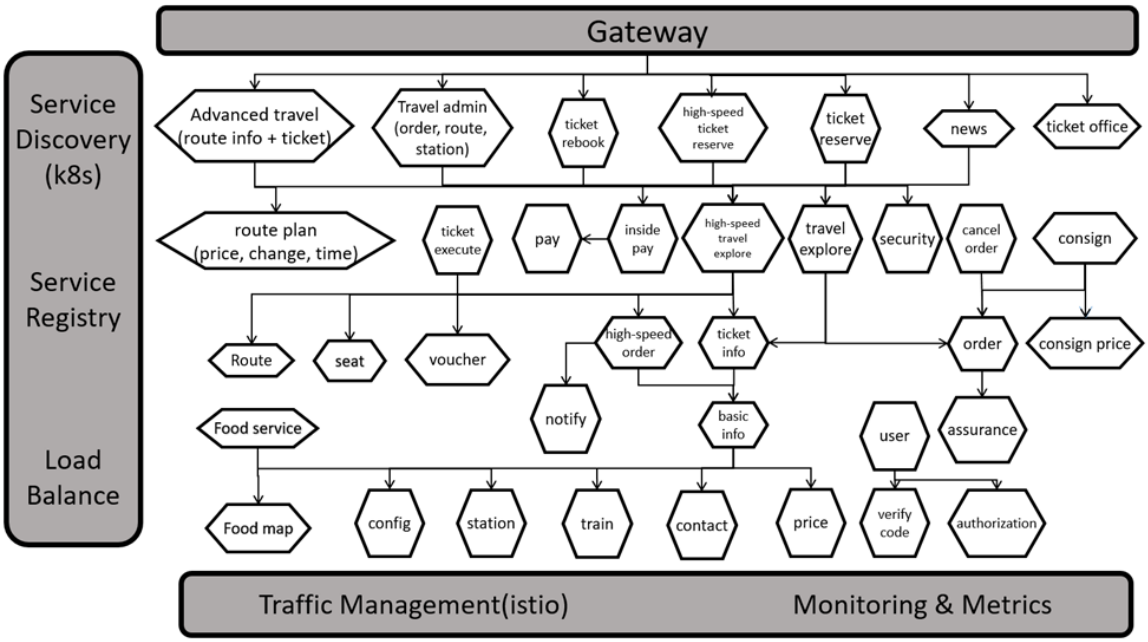
b. *The Opportunity*

Including caches between service calls requires effort from the developers, and then would need custom implementation for any such architecture. Since all of them go through the compilers eventually, we see an opportunity here of using compiler intelligence to identify points of communication between services as and when it is compiling the code, and then add caches on its own to achieve the same performance gains. Moreover, the compiler does not even require the source code, and uses the bytecode for analysis and addition of caches, which makes a helpful addition in sensitive environments when source code is not shared and only the bytecode is shared across servers running it. This can help multiple applications of the same language and application type to gain this advantage, also allowing the developers to focus on the application logic, rather than the caching logic.

There are obvious questions such as the passes being able identify the services correctly, the complexity and type of cache logic not suited for the application, the logic being language and compiler specific etc., however we believe that it would still help add caches to any applications using this compiler (implying the source code is not even required for this optimization), while also allowing the developers to not worry about adding this logic in the source code. While we start off with a simpler caching logic, with enough research and experimentation compilers can be made smart enough to answer all of the above questions and even provide flexibility to users.

*c. Tagret Application:*

The Train Ticket Application (link) is an existent high-performance application with nearly 50 microservices, as seen below:



Services such as ticket-reserve and route-info further interact with a chain of services, such as ticket-info, order etc., eventually falling into database requests such as config, station etc. The vast and frequent communication between services is visible here, and underlines the need of caches to improve performance.

### *d. The Approach:*

As this application is written in Springboot, a Java-based Open-source framework, we required the use of Soot, a Java-based Compiler Analysis Framework that also has inbuilt capabilities to analyze, modify and improve compiled bytecode before generating the binary. Soot provides capabilities to represent 3 Address Intermediate Representations that initially are output from the front-end parser and analyzer, in both Single-Static and non-Single Static form, called Jimple and Shimple respectively.

Our approach used this framework to write a compiler pass which would take as input the name of the service, then add cache calls onto places where there is RPC/REST API calls, and then generate the executable. For the scope of this project, we identified the service that made communicated regularly between other services, and also ensured that the nature of data in its requests are cache-able (i.e., is of a Key-Value type) and does not require immediate persistence or authentication to store in the cache (example, login data does not need to be cached). These are all assumptions we make since the scope of the project does not include intelligence in the compiler to support those parameters, however we will discuss these in the Future Work section.

### *e. Related Work:*

There has been plenty of research in bringing enough intelligence to compilers to introduce caches of some extent over RPCs, be it between two services, or between databases and backend software. We went over several papers as part of our literature survey, and decided to borrow some ideas from them while designing our pass. Ziv Scully et al. [1]'s work on adding compiler optimizations to cache SQL query results automatically to web applications without modifying the source code can give us sound ideas on how to create caches with concurrent access. Their work shown on many microbenchmarks promised a double or even higher increase in throughput, all with an extra argument passed to the compiler.

Amit et al [2]'s work on Database Scalability Services, which caches application's query results and answers queries on their behalf, also includes several aspects to consider when thinking about the possible scenarios where cache invalidation or staleness of data is a concern, while also encrypting the data for privacy reasons. Charlie et al [3]'s work on building a distributed cache can also give us insights on how to maintain coherence across the local caches of different microservices. The publish-subscribe system used in the Ferdinand Architecture is a viable design to borrow ideas from.

## 2. Details Regarding Your Design/Approach

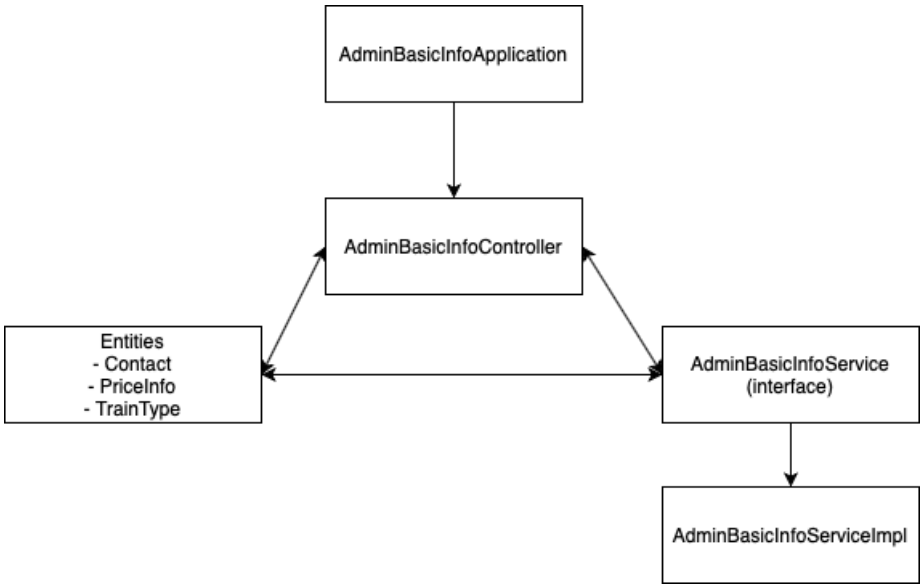
### *a. Initial Thoughts:*

As mentioned earlier, we analyzed the services and found that the ticket info service includes a lot of calls to other services, and also included a lot of data in its responses that was possible to cache, such as the list of trains between pairs of cities, or the schedule of trains based on their numbers. Analyzing the Source Code, we saw that it made use of the HttpRequestEntity and HttpResponseEntity classes in the SpringBoot libraries to format and package the data when sending through the network.

We then generated the bytecode for this service and noticed that the class names are labeled and present in the 3 Address-Code Intermediate Representation. We decided that our pass must then parse and detect these class names to identify REST/RPC calls. This brings an important assumption that the pass would only work for SpringBoot Applications, specifically those making use of these libraries, but could also be exposed as a parameter to the user in case the compiler is required to identify other classes or libraries.

### *b. Implementation:*

Once we started implementing, we saw that the train ticket application had a very complex design with a large number of RPC calls. To develop a PoC for our compiler pass, we picked the ts-admin-basic-info-service, in particular the getAllContacts() RPC call from this service. We identified that the call to method which has the actual REST call is present in the service implementation class. This service, like most others in Springboot had a typical MVC architecture. We have visualized it below -



The calls to ts-admin-basic-info-service are routed through the AdminBasicInfoController which then calls the interface AdminBasicInfoService to perform the calls. The implementation class for this has the actual RPC calls in the format -

```
ResponseEntity<Response> re = restTemplate.exchange(

            "http://ts-contacts-service:12347/api/v1/contactservice/contacts",

            HttpMethod.GET,

            requestEntity,

            Response.class);
```

We realized identifying and replacing this would be very tricky, as we did not have one specific value that we could use as our Cache Key in this call and we would have to come up with a composite key which contained the URL, HTTP request type etc.

So we decided to instead go one layer higher and identify the calling location for the service calls. These were present in the controller class and looked something like -

```
@CrossOrigin(origins = "*")

 @GetMapping(path = "/adminbasic/contacts")

 public HttpEntity getAllContacts(@RequestHeader HttpHeaders headers) {

    AdminBasicInfoController.LOGGER.info("[Admin Basic Info Service][Find All Contacts by admin ");

    return ok(adminBasicInfoService.getAllContacts(headers));

 }
```

This is a standard design for most spring boost based MVC applications so we decided to focus our attention here. We wanted to replace the “adminBasicInfoService.getAllContacts” call with our cache implementation.

*c. Approaches to identify RPC calls*

To identify this method call we explored the usage of Call Graphs in Soot. A Call Graph in Soot is a graph connecting all possible inter-procedural calls. Each node represents a method in the code base and directed each represents a possible invocation from the source method to the destination method. We found a few limitations with leveraging this for our use case -

1. By default, call graphs in soot only consider “main" methods as a possible entry point. This leads to a very limited number of edges being present in the call graph. When we generated the call graph for AdminBasicInfoApplication (which contains the main method) we could not even see the getAllContacts method as part of the call graph as there is no possible invocation path from main to getAllContacts. Since Springboot is responsible for starting the server and redirecting any calls to a specific endpoint to the corresponding handler, there is no explicit method invocation to these methods containing RPC calls.
2. We then tried to leverage the Soot provision to insert custom entry points into the call graph. However, since we have no information about which possible methods could contain the RPC calls, we iterated over every single method in every single class present in the application scope and added it as a custom entry point. This created an impossible large Call Graph, as it not only contained the entire train-ticket application but also all library method calls. We tried to parse through this to identify calls to the “restTemplate.exchange” method to identify location of RPC calls but it turned out impossible due to the extremely large size of the call graph.
3. We can take this approach forward by selecting a subset of method calls as entry points, by making some underlying assumptions considering the MVC architecture of the application. For example - only adding methods from the controller class. However, we decided not to spend more time on this approach.

As a second approach, we considered defining custom java annotations which the developer can use in their code to explicitly annotate RPC calls that should be optimized. In our compiler pass then we attempted to extract these annotations. However, before spending too much time on this we decided to not go forward with this as we wanted our compiler pass to not require any changes to the source code. Our goal was to have it work even if we had only the compiled class files in hand.

Finally we decided to go with a simple approach of passing command line arguments to our pass to specify the class name and method name which contains the RPC call and should be optimized. Although very basic we felt this works best for the current level of our implementation.

*d. Approaches to design the cache*

1. *Implement Cache in Java*

   This allows writing a generic cache class source code, and then using Soot to initialize an instance of this cache class in our target class, and then invoking the cache instructions and adding it to the service bytecode where the REST/RPC calls are made. This approach allows easy development of cache classes with different logic as desired as the Soot functionality here is to only insert the instructions at the right points. We follow this method.

2. Dynamically Inserting Cache Instructions

   This requires Soot to identify the data type of requests, create a cache data structure(map) of this type, and then insert the cache calls at the RPC/REST call points. Compared to the above approach, wherein the cache would be stored on the heap, the cache could be stored on the stack here (as local variables), and can bring faster access, although possibly insignificant. However, this brings a lot more complexity in the compiler pass since that needs to now figure out all of the above. Moreover, if we wanted to add different caching logic and protocols, writing in Soot is harder than writing in Java, since the JVM would take care of conversion.

   Initially we considered adding instructions to the IR to initialize a map, invoke methods to get/put from that cache and use that as our cache (approach 2 above). Although this was a good initial implementation, it would not scale as we would not have much room to build more mature caches with coherency protocols in place. Since java class loading is much more flexible than other languages that predate it, we leveraged this fact while building our pass. We wrote a java class with our Cache implementation which used Generics to represent the Key, Value types and implemented standard get/put methods. We then add this class also on our classpath.

e. *Design of the Cache Class*

In order to make the cache as generic as possible so that we are able to extend it for various types of applications, we decided on a simple Key-Value based Store. The Key and the Value types are generic, and allows storing any type of data that is sent across services. We defined something like -

```
public class Cache<K, V> {

  private Map<K, V> map;

  private int size;

  private final int CAPACITY;

  private int hitCount = 0;

  private int missCount = 0;

  public V get(K key);

  public void put(K key, V value);

}
```

Since ideally the compiler pass would add a cache at every point in the bytecode where an RPC/REST API call is present, there will be multiple caches present if the pass is applied onto multiple services. It is assumed that such caches do not clash with each other, and are independently able to modify data in the cache as they operate on their own instances of the Cache. We also assume that all caches are present on the RAM (no persistence), and that there is sufficient space allowed for multiple caches to co-exist. For the scope of this project, we only identify points of high frequency communication for inserting caches. We could set the size of cache as well as size of each entry as command line parameters.

Coming to the Cache logic itself, we realized a simple LRU cache would suffice i.e., to keep a counter for every incoming request that refers to the cache, and keep inserting entries as space available while also incrementing the counters for every request. Once we have no entries to fill, upon an incoming new request, we then check for the entry with the highest counter value, and then replace the same. We reset the counter of getting a cache hit for a request. This works well for temporal locality. To make the cache logic as generic as possible to serve most applications, we require implementing more caching logic, and then add compiler intelligence to select the most suitable logic based on certain heuristics/parameters of the application (this could also be

made known by the developers). We will leave this as a possible scope for improvement. We will also not use any associativity, since that would involve complexity in checking the addresses or data types.

We want to point here that although there should be logic present to ensure that before eviction, the data is safely sent across the network and persisted, we do not include that logic in our project, and assume that the data is already present in the service from which we are requesting data. In other words, we only cache requests wherein we read data, and not requests wherein we want to write data into.

### *f.* Adding the Caching logic through our compiler pass

To add this caching logic at the compiler level we identified we would have to do the following steps -

1. Declare a static field in the target class of type “Cache”

   We could do this fairly easily using the Soot APIs.

2. Initialize this static field in the target class constructor. If we have multiple RPC calls in the same target class that we have to optimize we must declare a separate field for each.

   To initialize the Cache class, we thought it would be a challenge as it uses Generics. When writing in source code, we need to pass the Wildcard type to initialize such types. However, in java when the code is compiled all these instances are replaced with the “java.lang.Object” type by default so we did not have to do this to initialize the field in our intermediate representation. But we did have to add special type casting instructions before we returned from our target method.

3. Replace the calls to the RPC calls with calls to our cache.

   To add the cache layer, we just add a wrapper over the function call that implements the REST API/RPC call, and then add this simple logic:

```
result = fetch_from_cache(request)

if (result == null)

      result = original_api_call

      add_to_cache(request, result)

return result
```

   This simple design ensures all calls must visit the cache first, avoiding calls maximally for data present in the cache.

### g. Implementing the compiler pass

Soot has the concept of Transforms which are basically a representation of a single compiler pass. There are two types of Transforms -

1. BodyTransform: Works on method bodies. Does not have access to any information from other classes/methods in the application. It is used for intra-procedural compiler passes.
2. SceneTransform: Works on the entire application and has access to all methods/classes on the classpath. It is used for inter-procedural compiler passes. This needs to run in what Soot calls the "whole program" mode.

As our compiler optimization requires inter-procedural analysis, we wrote our pass as a SceneTransform. After writing our transform, we simply need to add it to the list of transforms that Soot runs under the appropriate "Phase" and soot will run it as part of its compilation. At the end of execution, it outputs a new .class file containing the changes discussed above.

## 3. Experimental Setup

We used Soot v4.3(link), which is available as a jar file with all dependencies, and Java8 environment as our Base setup. Since we started with the Train Ticket Application, we required dependencies such as Docker and Maven, however as we found problems in deploying the application, we wrote a simple microservice application in Springboot which only required maven.

Since, our final execution was on our own user defined class to optimize inter-procedural calls we wrote the class Gateway.java to test our pass.

Our pass can be executed as -

```
javac -cp ./test test/Gateway.java test/Cache.java // compile test classes

javac -cp soot-4.3.0-jar-with-dependencies.jar:. Main.java CacheInsertionTransformer.java // compile the pass

java  -cp soot-4.3.0-jar-with-dependencies.jar:. Main ./test:. Gateway,Cache  // this will run the pass
```

Please check the README file in the repository for detailed instructions

## 4. Experimental Evaluation

We conducted many experiments to evaluate our pass -

1. Apply on the AdminBasicInfoController class from the train ticket application.

   We applied our pass on this service and were successfully able to generate intermediate representation in Jimple as well as output a .class file with the caching optimization in place. Unfortunately, we were unable to deploy the train ticket application to gather performance metrics on the optimizations achieved.

2. Apply on dummy microservices

   Due to issues with deployment of the above application, we decided to write our own dummy microservices that closely mirrored the design and architecture of the train-ticket application. We implemented 2 microservices where one microservice contains some data and the second one queries for that data through RPC calls. We were successfully able to apply the compiler pass on this application. We compared the intermediate representation of the output from our pass with the representation of the compiled code from actually making the changes to the source code and we were able to match it.

   However, when we tried to replace the .class generated by our pass to the .class files inside maven's target directory (which contains the built class files from "mvn install") the application failed to run. It gave a FileNotFound error for the target class, caused by a failure to parse the Configuration file which lists the application resources to be loaded. We spent quite some time on trying to debug this but we realized that it is much more complicated than we had initially thought. As maven also stores the compiled classes into its compiled jars and other resource files, replacing one simple class file breaks the maven dependency loading process possibly due to changes in the Project-Object Model.

3. Apply on inter-procedural calls within the same class

   Finally due the failure in above approaches, we decided to verify the working of our pass on inter-procedural calls made within the same class file. We wrote a simple java file where one method makes a call to another method. This invoked method could contain the RPC call or possibly any other expensive call like fetch from disk. We then used our pass to replace this method invocation with a cache and we were successfully able to see calls being made to our cache and calls to the invoked method reducing as cache hits increased. Although a very simple implementation, it proves the effectiveness of such a compiler pass which aims to optimize any kind of inter-procedural calls with a cache.

Since we don't have actual performance metrics due to the issues faced above, we would like to present a projected performance gain that could be achieved by such a compiler pass. Let's make the following assumptions -

Cache size of x where cache size represents the number of unique keys in the Cache at any time.

RPC calls follow the 80-20 rule (log distribution) where 80% of keys are only called 20% of the time and vice versa.

Total number of possible keys is y.

If x >= 20% of y, then we will get cache misses only in the first call giving an amortized RPC call tim complexity of O(1) for these 20% keys.

For remaining 80% keys, if they are equally likely to be called then we get a

Remaining cache size is x - 0.2y so we have a 0.8y/(x-0.2y) probability of getting a cache hit.

This would improve our overall performance significantly as we now have an O(1) amortized cost for 80% + 0.8y/(x-0.2y) % of our calls.

## 5. Surprises and Lessons Learned

We have discussed in depth the surprises we encountered while our development and how we changed and evolved our approach as we got over them. One of the main challenges we faced was the very low amount of documentation and resources on Soot. Additionally, a lot of features in Soot are still experimental (support for Java 11 is also currently experimental) and there is not much active development happening. When we looked for issues online we saw people from 2012 had created issues with the same problems that we were facing and there had been no work on it since then. There are also still some bugs in soot that we had to work around. Thankfully, the great open source community working on Soot are set to release a new overhauled version for Soot which should solve a lot of these problems and make it more user friendly.

We took away quite a lot in the form of lessons from this project. We focused too much on implementing our compiler pass in a clean and usable way and did not focus much on the actual practical implementations. Because of this we discovered the issues with inserting our own class files in maven projects very late in our project development and unfortunately could not dedicate more time to finding workarounds. Due to this error, we were unable to gather meaningful metrics around the real world performance of our pass.

## 6. Conclusions and Future Work

As we were able to successfully insert a Cache in the Microservice code, our analysis would be the same as that of the improvement achieved by inserting a cache in the source code, and the performance gains are dependent on the services that we chose to cache, as well as the type and size of the data cached. This brings a lot of promise in being able to smartly cache RPC calls in services without the source code being aware of this.

However, our implementation and experiments made a lot of assumptions (as stated above), and therefore leaves a lot of scope for improvement.

a. Improvements in the Pass:

First, our pass searches for specific names (such as HttpRequestEntity) that may be specific to frameworks (Springboot), and thus will not work on other Java Microservice Applications. We could either store such keywords for all commonly used Java Microservice Applications such as Springboot, Android, Tomcat etc and then identify the type of application based on some keywords, or allow the client to specify the keywords the compiler is to search and assume as points of network calls.

We must also include logic of estimating the size of the cache (per cache created), and the amount of RAM available, as excessive usage will lead to swapping by the OS, and cause further delays and complexities in maintaining consistency. The pass must first skim through the entire source code, then identify all the points of communication between services, and then conservatively add caches to them. If required to select a subset of such calls to be cached, Soot could then identify the frequency of making calls to different endpoints and then cache calls to endpoints with high frequency. For example, in the train ticket application, since the /ticket endpoint would be called the most across the application, it is helpful in caching all calls made to this endpoint.

b. Improvements in the Cache:
As the cache follows a simple expiry-based protocol, we need to include more variants of eviction and insertion logic, and allow clients to provide parameters of the application. For example, if the Application is more read-heavy, then spatial locality would greatly benefit, and so the pass must choose such a cache class. If the Application is more write-heavy, it would benefit from caching requests to log data into databases, and then push the data during eviction or invalidation. The cache logic written for these cases must ensure that data is eventually pushed and no data is lost.

We also do not include any associativity, and use First Come, First Serve entries which may cause capacity misses if the working set is too large. Since Soot could also attempt loop unrolling and loop invariant code motion to help improve Cache performance, independent of the Application.

If an application is heavily parallelized and the services are run on different cores of the same processor, then the compiler must ensure there is no staleness of data and coherency is maintained at all times. This would require MESI-like protocols, which introduce a lot of complexity to implement in Soot.

**References:**

[1] Ziv Scully, Adam Chlipala. A Program Optimization for Automatic Database Result Caching. Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'17). January 2017.

[2] A. Manjhi et al., "Invalidation Clues for Database Scalability Services," 2007 IEEE 23rd International Conference on Data Engineering, 2007, pp. 316-325, doi: 10.1109/ICDE.2007.367877.

[3] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. 2008. Scalable query result caching for web applications. Proc. VLDB Endow. 1, 1 (August 2008), 550–561. DOI:https://doi.org/10.14778/1453856.1453917

Online Resources:

1. https://github.com/soot-oss/soot/wiki/Tutorials
2. https://www.brics.dk/SootGuide/sootsurvivorsguide.pdf
3. https://github.com/soot-oss/soot/wiki/Class-loading-in-Soot
4. https://github.com/soot-oss/soot/wiki/Running-Soot

## 7. Distribution of Total Credit

Our total effort could be split into these parts:

a. Literature Survey
b. Soot Installation, Logistics and Experimentation
c. Installing and Deploying Train Ticket, Writing Another Sample SpringBoot Application
d. Writing Cache Source
e. Writing Soot Compiler Pass

Initially, we had split effort i.e. Adarsh completed Literature Survey while Bhakti finished all Soot related logistics, however all other tasks were done together, and so our effort was approximately evenly (50-50) split.