Inserting Caches using Compiler Passes to Optimize RPCs between Microservices

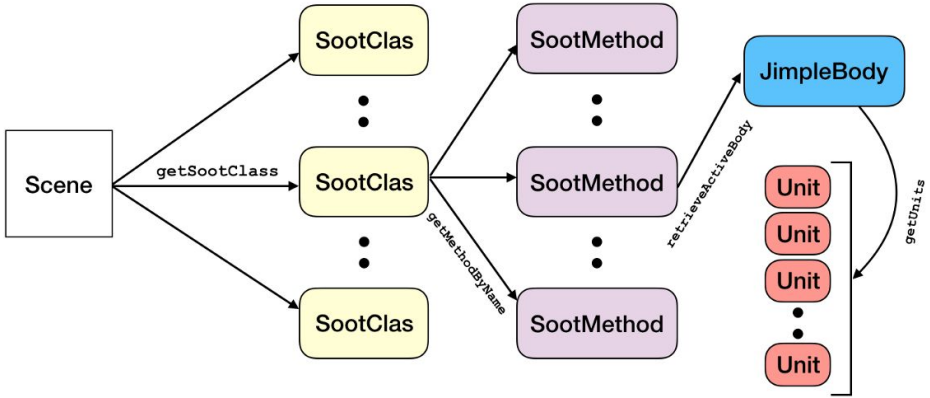
Adarsh Sreedhar
Bhakti Chaudhari

Overview

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. But with the simplicity of independently configurable and deployable unit comes the trade off of increased communication cost. This introduces an opportunity for caching these RPC calls, especially when they are read-only or read-mostly calls. Since it is a tedious and repetitive burden on the developer to write this caching logic in their source code, we see an opportunity to leverage compiler level optimizations to do this transformation. Here, we try to optimise java SpringBoot microservice applications using a compiler pass to insert a cache to replace RPC calls.

The Approach

- Leverage Soot to design the compiler pass. Soot is a Compiler Optimization Framework to visualize, analyze and transform Java Applications.
- Use the Soot APIs to tweak existing microservice code to add new fields, instantiate objects and add method invocations for the chosen Cache implementation.
- Generate and analyze the inter-procedural Call Graph to identify possible optimization points by flagging RPC calls.



The Design

- Design a Cache class in Java leveraging Generics for type information. Implement your desired cache protocol.
- Develop a compiler pass which is able to identify possible optimisation points by recognising RPC calls.
- Insert and instantiate the Cache object in the target class.
- Apply transformations to substitute these calls to a call to our Cache class. If there is a cache miss then insert back into the cache.

The Implementation

- We implemented this pass by analysing one of the intermediate representation forms used in Soot called Jimple.
- 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
```

The Transform

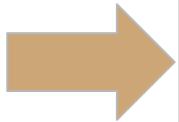
- The transform was implemented as part of Soot's SceneTranform phase which is responsible for conducting inter-procedural analysis.

```
public static java.lang.String getInfo(java.lang.String)
{
    java.lang.String r0, $r1;

    r0 := @parameter0: java.lang.String;

    $r1 = staticinvoke <Gateway: java.lang.String getInfoCall(java.lang.String)>(r0);

    return $r1;
}
```



```
public java.lang.String getInfo(java.lang.String)
{
    Gateway r3;
    java.lang.String r0, $r1, $r2, $r4;
    com.example.gateway.Gateway $r5;
    Cache cacheLocal;

    r3 := @this: Gateway;
    r0 := @parameter0: java.lang.String;
    $r1 = <com.example.gateway.Gateway: java.lang.String url>;
    $r5 = (com.example.gateway.Gateway) r3;
    cacheLocal = <Gateway: Cache cache>;
    $r4 = virtualinvoke cacheLocal.<Cache: java.lang.Object get(java.lang.Object)>($r1);

    if null != $r4 goto label2;

    if null == $r4 goto label1;

label1:
    $r4 = virtualinvoke $r5.<com.example.gateway.Gateway: java.lang.String getInfoCall(java.lang.String)>($r2);
    virtualinvoke cacheLocal.<Cache: void put(java.lang.Object,java.lang.Object)>($r2, $r4);

label2:
    $r4 = (java.lang.String) $r4;
    return $r4;
}
```

Results

- The bytecode generated by applying the pass is the same as the byte generated by compiling Source with Cache added.
- We applied our pass on a test file to optimize an inter-procedural call within methods. As we tested with a sample data size smaller than the cache capacity, we got an amortized hit rate of 100%.
- When testing with a cache capacity x% of the sample data size and generating randomised requests, we were able to get an approximate hit rate of x% following a linear pattern as expected. The cache protocol was LRU.

Limitations

- Difficult to generalise the pass as different RPC frameworks use different patterns, designs and underlying libraries.
- Need to analyse correct places to apply this pass as we do not want to introduce bugs due to stale data.
- Need a generalisable way to identify possible optimization sites that is generic enough for most java applications.
- Difficult to implement as part of a larger Springboot or similar projects as it has complex dependency management which needs to be accounted for.
- Dealing with cache coherency for data which may change over time.

Future Scope

- Extending the pass to apply to different types of HTTP and RPC frameworks in Java.
- Implementing mature cache coherence protocols like MESI.
- Adding support for additional types of Caches like LRU, LFU, expiry based etc.
- Using heuristics and dynamic program analysis to pick a ideal caching protocol, cache size etc.

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