Compiler optimizations for micro-service based Cloud Applications

Sanil Rao*, Pil Jae Jang*, Pratik Fegade†, Todd Mowry†
*Carnegie Mellon University, Electrical and Computer Engineering
{sanilr,piljaej}@andrew.cmu.edu
†Carnegie Mellon University, School of Computer Science
{ppf,tcm}@cs.cmu.edu

INTRODUCTION

Since its inception, the cloud has been an integral part of many modern businesses. The cloud allows corporations to scale out their applications without having to spend large sums of money acquiring all the underlying hardware. As such, many popular applications have moved from locally hosted to cloud hosted using one of the large cloud providers.

When moving to a cloud environment the application itself also has to change. It is no longer ideal to have it as a giant monolithic application. Instead, the application should be broken up into smaller micro-services that communicate with one another. This reconfiguration better fits the underlying structure of modern cloud computing centers. From a compiler perspective these new micro-service applications provide an interesting challenge for optimization. Given how novel this application design approach is, not much time has been spent on exposing optimizations to the compiler. For example, these micro-services are constantly communicating with one another, which could incur large overheads. Optimizing services that talk frequently to each other could minimize computation time, giving increased cost savings.

The goal of our project is to implement a compiler optimization that finds high communication between different micro-services and minimize this communication cost at run time. We look at two specific optimizations, communication on shared-memory machines, and loop-based communication over a set of values. With shared-memory machines, rather than communication via the network, you can access the data directly. Our optimization transforms this type of communication to used shared memory instead of the network. With loop-based communication, we wanted to remove repeating network communication with a single bulk call of all the data.

We were pointed to a few resources by Professor Mowry for this project. One was a talk given by a compiler group from IBM that discussed the compiler framework we would be using. Its design is tailored for micro-service based applications. Another is the blogs of some the OMR developers. They had key insights on how to get started and were critical to the design of our compiler optimizations.

The contributions for our project is that we were successfully able to create optimizations that reduce of the overhead of network communication seen in many micro-service applications. We implemented two specific optimizations, utilizing shared memory and aggregating communication. While we used toy examples for our project, one can apply what we have utilized to production applications that use the OMR framework, like those applications run in the OpenJ9 JVM.

APPROACH

ACME Air and OpenJ9

ACME Air [1] is a production micro-service application built alongside the OpenJ9 JVM [2]. This application simulates an imaginary airline with bookings, tickets, flight logs etc. This was the initial target application for our project, as it had many areas that we could optimize. It also worked very well inside the OpenJ9 JVM exposing many of its capabilities that we could take advantage of. However, we were unable to get these two pieces to work properly during development. While we were able to run both these pieces in the IBM cloud, we had no control over the actual implementation on our local machine, therefore we could not modify them to include our pass.

We then switched to the monolithic version of ACME Air to run on our local machine. Unfortunately, this was also unsuccessful as some of the components in the install guide were not working as originally written. Given the amount of time sunk we decided to transition to c++ synthetic applications for the remainder of the project. Our synthetic applications mimic some of the same functions seen in this application so in time one can employ our optimizations there. Another area that we believed would be of concern is how to inject our code within the OpenJ9 JVM. OMR is built within JVM itself so it was unclear during development how to properly modify this to our needs. Given more time we believe we could solve these issues.

OMR JIT Overview

In order to complete our project we first had to understand the environment that we were using, OMR [3]. This is the compiler framework that we used to develop our pass as the applications we originally planned to target were built with this framework. OMR is designed specifically for JIT applications, and exposes in-demand JIT features to developers. The basic steps of the OMR JIT process is as follows; first you have to initialize the JIT component of the framework. After that you

have to define a type dictionary and a method builder. The type dictionary, serves as a way to get the JIT to interface with the types used in the interpreter. The method builder, contains the core implementation of the JIT's functionality. Finally when you want to invoke your compiled function you simply create a call type that the JIT would recognize, replacing the original function with its compiled version.

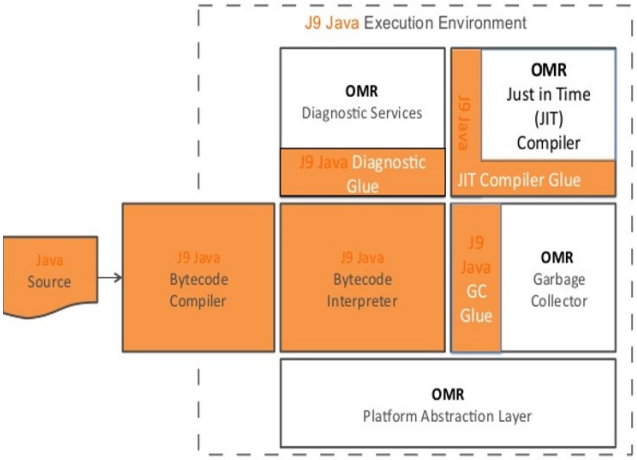

Fig. 1. OMR framework Overview in Java Runtime System

OMR Intermediate Language

Having an understanding of the JIT structure we moved on to the component that we wanted to modify, the Intermediate Language(IL). IL is similar to the Intermediate Representation(IR) in LLVM, and serves exactly the same function. Unlike LLVM however, the OMR framework has no frontend component. This means that there is no way to take a program and gets its translated IL, like you can in LLVM. Therefore, one has to write for each compiled function defined above, the IL for that function. While not entirely difficult, this became a bottleneck when it comes to testing intricate functions, as we had to write the function twice, in the frontend language and in IL. Furthermore, the IL doesn't have all the same types as C++. For example, strings cannot be represented in IL directly, rather you have to take the address of the object and do some type conversions to get the final string to show up. In hindsight the fix wasn't very difficult to implement, however at the time it took many hours to understand.

The structure of the IL has some similarities and differences when compared to traditional compilers like LLVM. The underlying representation of the IL is a series of Directed Acyclic Graphs(DAGs) [4], which contain an operation and yields a result. Within a DAG is a node that has the operator, and its children have the operands. These DAGs can have side effects [5] which determine ordering between nodes not expressed through edges in the DAG. In this instance a linked-list structure called a TreeTop traverses the DAG to determine program order. Anything reachable from a TreeTop is called a tree, which looks like a DAG itself, and anything reachable from a node is called a subtree. For the purposes of our project we thought of this as Trees being basic blocks and nodes being instructions. While not entirely accurate it was enough to get our passes to work properly.

```
a = b + c
b = a - d
c = b + c
d = a - d
```

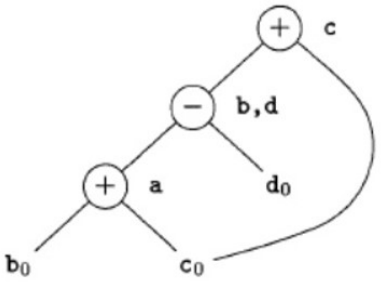

Fig. 2. DAG example

Optimization Pass Overview

Having understood the major components of the framework and the language, the next step is actually implementing our pass to improve our application. We decided to build upon some of the existing passes that were written in the OMR framework. Because OMR is a JIT framework, optimization works differently than traditional compilers like LLVM; they are applied a few times during program execution rather than just once. Optimization comes in tiers starting from cold (low) to scorching (aggressive). Each tier determines which types of optimizations are performed, directly affecting compile time. We chose to work with the hot tier which is the middle level but made sure to specify our unique pass run first before other passes in the hot tier.

Shared Memory Optimization

Our passes extend from the given optimization DeadTreeElimination, colloquially known as DeadCodeElimination. For the shared memory optimization pass we first make the assumptions that the two micro-services are on the same machine and can access the same address space. If either of these do not hold our pass will not work. Our pass looks through all the Trees in the function statically to determine if within a Tree there exists a node that makes a function call between the two services. If such a call exists, we know thanks to our assumptions, that service A already has the data it requests from service B. Therefore we remove the offending call to service B by eliminating the node (instruction). We then create an instruction with an access to the object's address in memory to service A which it uses for the rest of program execution.

Loop Aggregation Optimization

For the loop-aggregation pass we have the assumption that service B understands how to parse a new aggregate procedure call. In order to do this we had to run two different passes on each of our services A and B. For service A we had to modify the loop'd procedure call to be a single bulk procedure call containing all data being loop'd over. This was done by first surrounding the loop with a never taken branch. This

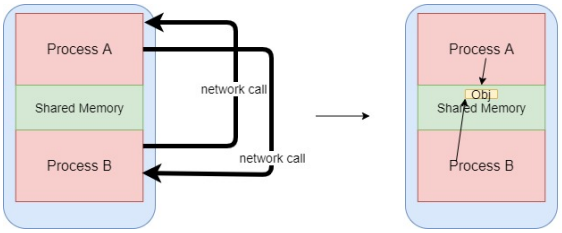

Fig. 3. Shared memory optimization visualized

allows DeadTreeElimination to remove the loop entirely from the IL. Then a new procedure call was created, sending the objects' full contents over as the data. The full contents was represented as a single meta object composed of each object contents concatenated together with a delimiter in between them.

Meanwhile for service B we had to create a pass that modified how to parse the data being given in order to gather all data from service A. This meant parsing the meta object by the delimited and the zipping it up again in a fashion similar to that described above. This split pass process needed to be done because we had separate source files for each service with separate IL.

In reality this pass style does not need to be done as most applications have a single driver that would be able to access both services, so it would be able to combine the 2 passes into 1 pass.

Experimental Setup

We ran experiments that showed the performance and compilation time of the original and optimized code. To run these one first has to download the omr framework from github. Then during the build process turn on the jitbuilder extension as that is the runtime system we were working within. After this is complete you can run some of the example programs in the provided folder using the standard make and execute for normal C++ programs. Additional information can be found at [6] blog post.

For our experiments we extended the DeadTreeEliminiation file found in the optimizer to perform both the shared memory and the loop aggregation optimizations. We created a build script that would recompile the jitbuilder with our new optimization. Then we compile the source program and use the new jitbuilder runtime during program execution. We specify in the execution command to run our optimization first as that is what we are most interested in and we did not want interference from other passes. We time the compilation process using the bash timing functions. We time program execution using the c time class. For our validity analysis we use the same timing procedures for both the original and the optimized code.

Experimental Evaluation

Figures 5 and 6 show the results of our optimizations on both performance and compilation time respectively.

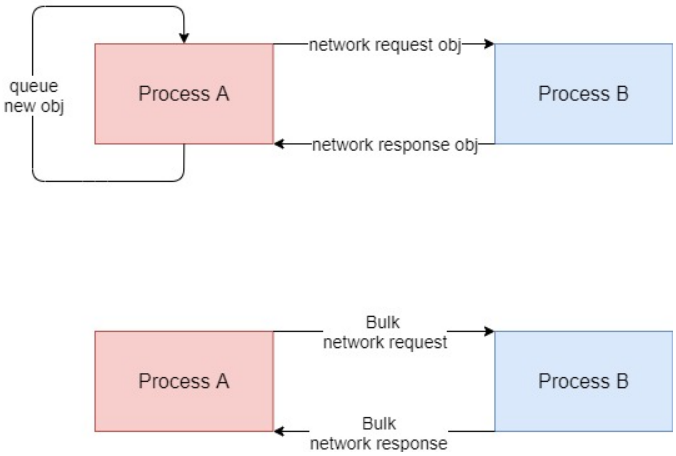

Fig. 4. Loop Optimization Visualized

It is clear that the optimizations we performed had an improvement in execution time. We were able to see speedups of **1.4x** for the shared memory optimization and **1.9x** for the loop aggregation optimization. Accessing shared memory is much faster than using a socket for communication. Thus it makes sense that there would be an improvement when transforming this communication; this is corroborated empirically in this paper [7]. The key reason this works is because the driver program spawns off these processes at start up. If instead one had to start these processes separately more work would have to be put in to make the data structures accessible to each service. One issue with this experiment is that our "network" was actually all run on a single system. Therefore, in reality we would see much better performance if we actually had to move from one machine to another machine.

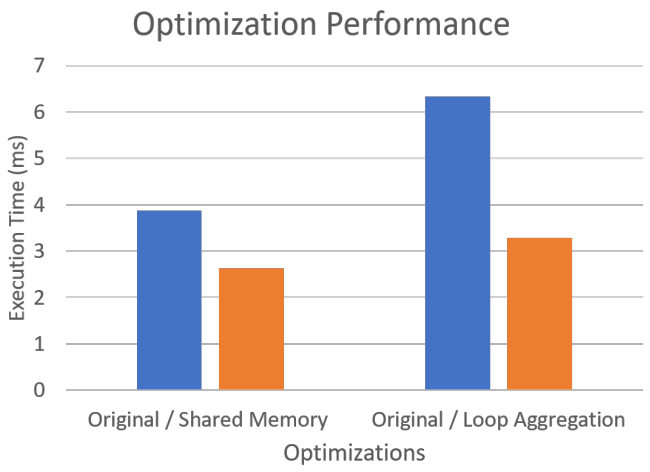

Fig. 5. Execution time of the original and optimized versions of the program. Clear decrease in execution time for both optimizations

As for the loop aggregation optimization reducing the volume of network calls would obviously reduce the amount of time for program execution. This is because less time is

spent waiting on data to be sent back and forth between services. That is why we see such a substantial performance improvement. We believe this improvement would be even greater if we ran our experiment using multiple machines as the latency for each network request would be that much larger. One issue with this experiment is a scenario in which there is more computational expensive tasks being performed between the two services. In this case if we aggregate and send it might take even longer to get a response back. Furthermore, if in the original case there was some parallelism between requests, then we have effectively removed this. Now process A sends all the data so it has to wait for all the data whereas before it could continue on each element as it came in. Testing this scenario would add further validity to this optimization and is left for future work.

In compilation time we didn't see too big a difference between the original and the optimized versions. We attribute this to the fact that this was mostly a proof of concept work. Our passes were not as robust as they could be and are limited in their power. A few assumptions were baked in as well in order to get a working pass. We don't in fact believe this to hold if someone were to go in and create a generalized version of the passes we have designed here. They would add an additional amount of time to compilation as more work would need to be performed. As an example conversion of the network request to take bulk rather than single element data is not a trivial operation, especially for more complex object types.

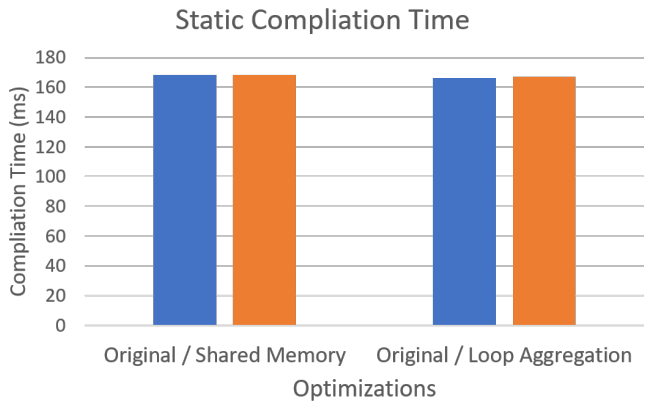

Fig. 6. Compluation time of the original and optimized versions of the program. Negligible impact on compile time with the introduction of these passes

Additionally, Figure 7 illustrates our validity analysis. This validity analysis was a manual transformation of some more complex programming examples. These were taken from real world microservice applications to see what the benefit would be. It is clear that there would be a significant improvement employing our passes on these programs. We see speedups of **1.57x** and **1.99x**. This further demonstrates the effectiveness of these passes.

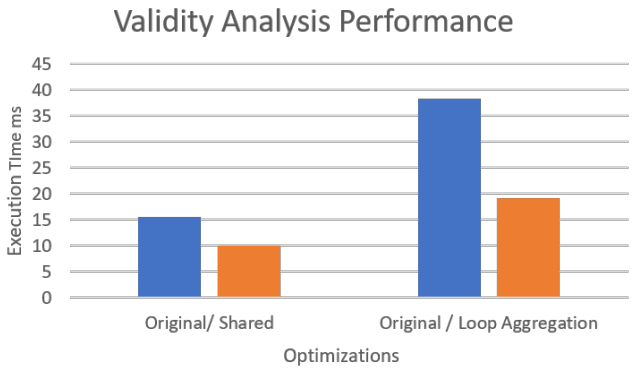

Fig. 7. Validity analysis showing our pass on real world programs. Highlights the power of these types of optimizations on performance

SURPRISES

Overall nothing about our work was that surprising or counter-intuitive. We believed at the start that our optimizations would yield a benefit and as it turned out they did so. We believe the results will be even better than we show when tested in a production system. We were saddened we were unable to use a production application form the get go. This was due to the inability to get the application to run properly locally as well as the the start up time to understand the framework. It is quite difficult to jump into a new software system without much documentation. Even simple operations take quite a while to get the hang of in a new environment. Many times our communication with the OMR developers involved sending snippets of LLVM passes and asking for an idea of how to translate these to their system. This did give a dos and dont's list when it comes to trying to learn new software systems, which will be invaluable in the future.

CONCLUSIONS AND FUTURE WORK

In this paper we have created two compiler optimizations to improve microservice applications. Our proof of concept passes aim to reduce the cost of performing network communication between two micro services by either utilizing the underlying shared memory of the machine or aggregating the requests into one large request. We show that for our example program we see a benefit for both our passes with little effects to overall compile time. We then perform a validity analysis to provide additional examples and strengthen our claims.

In future work we look to test these passes on more advanced or real world applications. In addition we hope to generalize these passes such that they can be used out of the box on a variety of programs. This is important for those who wish to use these in the future. Finally, we hope to implement additional passes that aim at the goal of reducing or eliminating network requests in a program, a major bottleneck in these microservice applications.

REFERENCES

[1] IBM, "Acme air sample and benchmark." https://github.com/acmeair/acmeair.

[2] M. Stoodley, "Omr a modern toolkit for building language runtimes." https://www.slideshare.net/MarkStoodley/omr-a-modern-toolkit-for-building-language-runtimes.

[3] M. Stoodley, "wasmjit-omr jit compiler." https://github.com/mstoodle/wasmjit-omr/tree/master/src/jit.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[5] IBM, "Testarossa's intermediate language: An intro to trees." https://github.com/eclipse/omr/blob/master/doc/compiler/il/IntroToTrees.md.

[6] M. Stoodley, "Building omr jitbuilder with cmake." https://mstoodle.github.io/BuildingJitBuilderWithCMake/.

[7] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms," *Architecture*, vol. 86, p. 64, 2015.

DISTRIBUTION OF TOTAL CREDIT

Sanil Rao 50%

Pil Jae Jang 50%

PROJECT WEBPAGE

https://piljaej.github.io/15745_project/index.html