JVM Footprint reduction for Edge Devices

Alexandra Tsvetkova, Bojian Zheng, Marius Pirvu, Gennady Pekhimenko
Course: CSC2228 Advanced Topics in Mobile and Pervasive Computing: Edge Computing
Fall 2019

***Abstract*— This project is aimed at studying opportunities memory compression can give at the Edge environment. Using Raspberry Pi 3 we are going to get real life data from Java applications to evaluate compression ratio and performance overhead of Java applications on edge devices.**

I. Project description

A. Original project

Java Virtual Machine (JVM) is an engine that provides runtime environment to execute java applications in a safe and controlled environment, managing not only instruction execution but also memory allocation and usage. This environment is especially popular because it allows to run applications on different types of devices. Write-once, run-everywhere concept, supported by Java, is very convenient for application developers.

A new research direction IBM is looking into is called ”continuous platform experience”[4]. Running JVM based applications in the cloud and mobile devices can provide a unique user experience. Unfortunately, memory is the most expensive resource in the cloud (see Table I [5]). Memory on the edge devices very limited as well.

To store and keep track of its objects JVM uses heap of limited size. This could become a severe bottleneck for execution of large application loads, especially in cloud or datacenter ecosystems where low memory consumption could allow for higher execution node occupancy.

Instance Type	vCPU	Memory	Price
t3.nano	2	0.5	0.024
t3.micro	2	1	0.0479
t3.small	2	2	0.0958
t3.medium	2	4	0.1917
t3.large	2	8	0.3834

TABLE I
Amazon EC2 Linux pricing

At the same time, Java objects could be compressed with a high degree of efficiency to allow for the reduced memory usage. In our experiments we compressed the heap using gzip algorithm and achieved a compression ratio of 7.19. Of course, any potential memory optimizations are significantly less attractive if the execution overhead is too high.

We propose an implementation where compression will be done within the garbage collector (GC). We add the compression stage at the end of garbage collection. Most of the overhead here can be avoided by executing compression in a parallel thread, together with the garbage collector. We are planning to perform compression of the tenured area (i.e. only for objects which are present in memory for reasonably long amount of time). Another avenue for optimization is to perform compression only on cold objects - the challenge of that approach lies in discovering the cold objects in the first place.

Another crucial factor is memory decompression latency, since it can have a severe impact on memory access performance. To address this issue we aim to implement a low latency decompression algorithm. The decompression code will be injected within the bytecode of the user application.

B. Edge computing

Typically, edge devices are resource constrained[1]. Compressing memory on the device can help us trade off compute resources for memory. Unlike desktop machines, both these resources in the Raspberry Pi are very limited. It will be interesting to study the trade off for edge devices.

C. *Previous research*

The previous research [2] showed high compressibility ob java heap. It showed that application can be effectively reduced by up to 86%, on average 58%.

The major benchmark we are planning to use in this research is DaCapo[3] because it consists of a set of open source, real world applications with non-trivial memory loads. we am also planning to use SpecJBB2005.

II. Current progress

A. *Raspberry PI setup*

In this experiment we used Raspberry PI 3B+ with Ubuntu Mate 18.04. While setting up the environment, we encountered several difficulties:

1) Setting up ssh to Raspberry Pi. Ssh connection from my desktop machine (connected to the university network via cable) to the Raspberry PI (connected to compsci Wi-Fi network + VPN) doesn't work. We haven't found the reason for that yet. For now we are using Raspberry PI with a monitor, but it's inconvenient. An interesting fact is that we can ssh from Raspberry Pi to my desktop and we have tried to do reversed ssh, but the connection gets dropped.
2) There is no OpenJ9 version for ARM. OpenJ9 exists for x64, s390x and ppc64le, but we couldn't find a version for arm. We did the testing with OpenJDK8. The performance results should be similar to OpenJ9, but there are several difficulties:
    - For my microbenchmarks we used to exclude main function from the compilation to avoid inlining of the instrumented code into main. It affects 2 microbenchmarks where we are doing summation of the array elements (RandSum and ZeroSum)
    - we used various JIT compiler options to do scorching compilation early (to make sure that compilation is not happening during time measurements) and to get heap dumps. We are still looking for the appropriate options for OpenJDK8 runtime.

B. *Loading array elements*

When we were fixing the instrumentation, we have noticed that for primitive types instead of executing the AALOAD instruction for every array elenemt, the runtime executes ALOAD. The difference between these two commands is that AALOAD keeps the reference to the head of the array and the element index on stack. The ALOAD only keeps the final reference on stack [6]. Because of this issue, it is really hard to get access to the array head from the instrumentation. It will make it hard to do compression/decompression for the arrays of primitive types.

For this experiments I used a simple check whether the array element is 0 or not (making a load from the address in the instrumentation), but it is not a good solution for compression because we don't have access to other elements.

We are thinking of different ways of solving this problem. First, we have to decide whether it's better to do page-based or object-based compression. To make that decision, we will need to analyse heaps of our the most popular benchmarks and extract per object information from it. Second, we have an idea of switching from bytecode injection to write barriers based decompression. We still need to estimate overhead of that method.

C. *Performance results*

In Fig 1 we can see that the performance overhead is very different for different microbench-

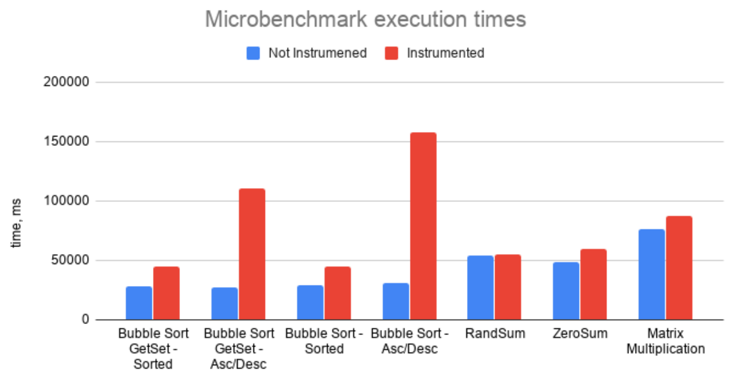

Fig. 1. Performance results of the microbenchmarks on a Raspberry PI

marks. For most of the microbenchmarks it stays within 50%, but for Bubble sort with flipping the array it can reach up to 5x.

III. Updated Timelines

This project consists of multiple parts:

Part	Timeline
Set up Raspberry Pi without IBM J9	DONE
Fix up decompression code - support array elements	DONE
Main project poster on CASCON	DONE
Get Microbenchmark Performance results on Raspberry Pi	DONE
Get heap dumps from Raspberry Pi Apps	End of Nov
Get final Performance results on Raspberry Pi	Dec
Switch from bytecode injection to write barriers	Mid Jan
Find a place in the GC to store metadata	Mid Feb
Implement compression in GC	End of Feb
Get final performance data	End of Mar
Add Hot/Cold objects	TBD

References

[1] Dependability in Edge Computing. Paul Wood, Heng Zhang, Muhammad-Bilal Siddiqui, Saurabh Bagchi

[2] No Bit Left Behind:The Limits of Heap Data Compression. Jennifer B. Sartor, Martin Hirzel, Kathryn S. McKinley

[3] The DaCapo Benchmarks:Java Benchmarking Development and Analysis. Stephen M Blackburn et al.

[4] `https://www.infoworld.com/article/2614812/ibm-explores-making-java-virtual-machine-big-part-of-future-cloud-platforms.html`

[5] `https://www.amazonaws.cn/en/ec2/pricing/ec2-linux-pricing/`

[6] Wikipedia, Java bytecode instruction listings