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 that 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. Introduction

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.

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.

Java objects are initially created in the Eden region (see Fig. 1). If this region is full, garbage collector (GC) starts working. GC marks all the objects on the Young Generation of objects (Eden + Survivor). Half of the Survivor space is always kept empty (To region). It is used for the surviving objects during garbage collection. After all the living objects are moved to the To region, the rest of the Young generation is wiped out. The long living objects are moved to the Old Generation (Tenured region). It is not reasonable to compress young generation because it mostly contains short-living objects with high probability of death within a few next GCs.

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

We propose an implementation where compression will be done within the garbage collector. 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.

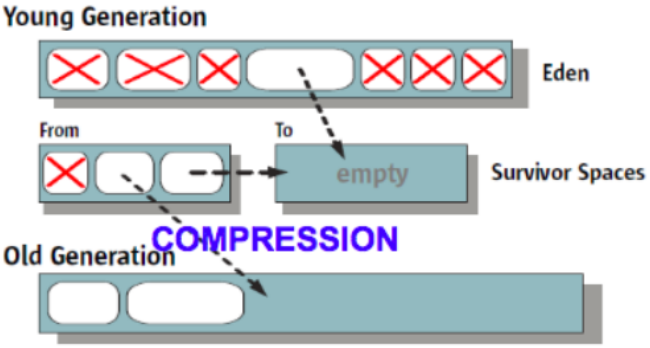

Fig. 1. Compression in the object lifecycle

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]. Table II demonstrates memory constraints for Raspberry PI devices. 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.

	Model A	Model B	Model A+	Model B+	Pi 2/3	Pi Zero
SDRAM	256 MiB	256 MiB / 512 MiB	256 MiB	512 MiB	1024 MiB	512 MiB

TABLE II
RASPBERRY PI MEMORY

II. EVALUATION

A. Experiment setup

In this experiment we used Raspberry PI 3B+ with Ubuntu Mate 18.04. I has a 1.4GHz 64-bit quad-core processor and 1GB LPDDR2 SDRAM. OpenJ9 is only available for x64, s390x and ppc64le platforms, and we were unable find a version for ARM. We performed our experiments with OpenJDK8.

B. Implementation details

For this project we implemented a bytecode injector in order to simulate decompression latency. The expectation was to get reasonable decompression overhead of 20-30%, which is a reasonable overhead for real-life heap compression usage.

When JVM starts executing an application, all the files containing bytecode of each class get loaded in the JVM. This is the moment injection happens. Our bytecode injector modifies the class bytecode. For every load and store it duplicates the stack top (the address where memory call will be happening) and adds a function call on top. This function takes exactly one argument (the address that we duplicated) and analyses the data stored there. During the function call, extra value from the top of the stack gets popped into the function. This leaves the final stack in the same state as before.

Later on bytecode in the hot/warm regions gets compiled for the target architecture. Java runtime determines the most frequently executed functions and after a number of executions it starts compilation with optimizations. Our injected code will be compiled in line with the initial code of the class. These optimizations play a key role in java code performance. As we can see from the evaluation, injection of a couple lines of bytecode can lead to completely different optimization paths and overhead results which are hard to predict in advance.

For our evaluation we used a simple function with a single ”if” statement and 2 counters: global counter of all function calls and a counter of zero function calls. This simulates the most trivial case of decompression - zero decompression. We will not be able to make decompression cheaper than a couple cycles because it will take at least one if statement per memory access (checking if the object is compressed of not).

C. Microbenchmark results

We implemented a set of microbenchmarks to evaluate bytecode injection overhead.

First four microbenchmarks are simply evaluating overhead of bubble sort in 4 different cases: instrumenting the whole function vs instrumenting only setters and getters for array elements and sorting already sorted array (n^2 checks) vs flipping it every time.

We tried to implement similar microbenchmarks for the quick sort, but the results were very unstable (up to 70% difference on different runs) because of the recursive nature of this sorting and JVM runtime optimizations).

Matrix multiplication is simply a benchmark implementing multiplication of 2 large matrices.

We have 2 more benchmarks for summation of huge arrays (1 Gb array of integers). One benchmark sums up random numbers, another one sums up an array of zeros.

For all microbenchmarks we disabled compilation of main function (to prevent inlining of the evaluated functions). We also make sure to pre-heat applications to make sure that evaluated functions reach the hottest compilation level before starting the measurements.

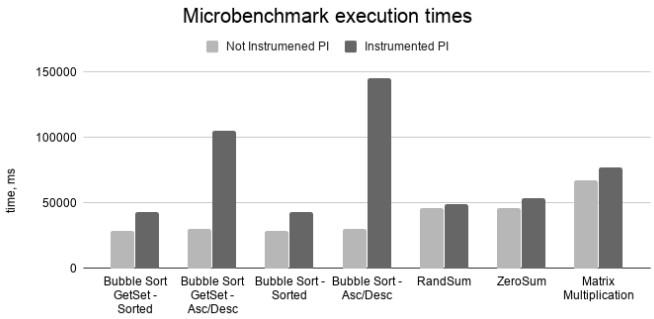

Fig. 2. Performance results of the microbenchmarks on a Raspberry PI

In Fig. 2 we can see that the performance overhead is very different for different microbenchmarks. For most of the microbenchmarks it stays within 50%, but for Bubble sort with flipping the array it can reach up to 5x.

One interesting result that is different on server machine and raspberry PI is bubble sort benchmark behavior. On raspberry PI we get high overhead when we flip the array, on powerful server machine it’s other way round - we get higher overhead in the sorted case.

Overall, the overhead is quite affordable in most cases. We are still looking at the opportunities of reducing overhead.

D. Benchmark results

Fig. 3 shows us the bandwidth degradation for instrumented version of Spec Jbb 2005. In this benchmark multiple warehouses (different threads) attempt to perform transactions at the same time. The resulting throughput shows aggregated transactions from all warehouses.

For 1 warehouse the throughput degradation is around 30%, but we can see that this throughput doesn’t increase

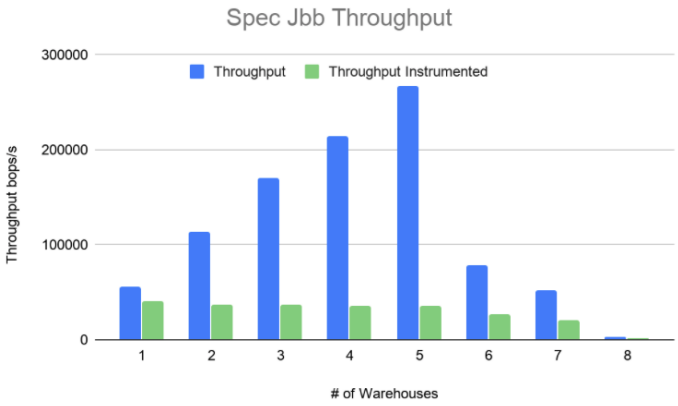

Fig. 3. Performance results of the ScpecJbb2005 on a Raspberry PI

with the number of warehouses if we use instrumentation. This result is surprising and it can't be explained as of now. We can see that the number of busy cores (over 90% load) grows with the number of warehouses in both cases.

For non-instrumented SpecJbb we can see a significant drop after 5 warehouses because of the frequent Garbage collections. All warehouses are using the same heap of limited size and they start running out of memory.

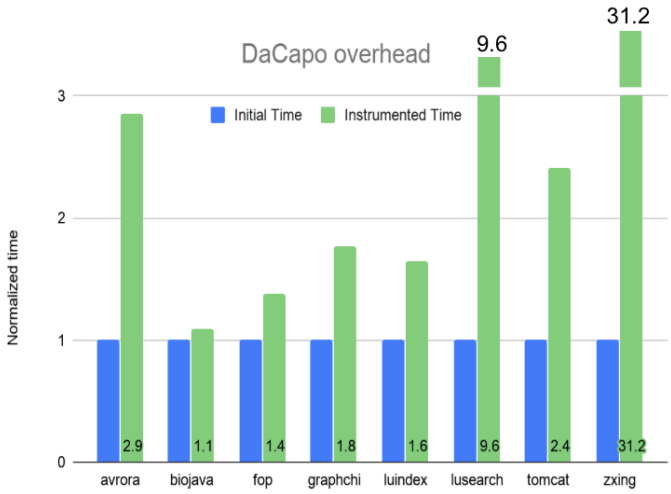

Fig. 4. Performance results of the DaCapo benchmark suite on a Raspberry PI

Fig. 4 demonstrates results for the Beta version of DaCapo benchmark suite. It represents application with various models of memory usage. We have achieved high variance in results for different benchmarks, but we believe that it happened due to unstability of the benchmark suite. We have observed crashes on 11 out of 19 benchmarks even with the default JVM.

E. Compressibility

$$Compressibility = \frac{Initialdata}{Compresseddata}$$

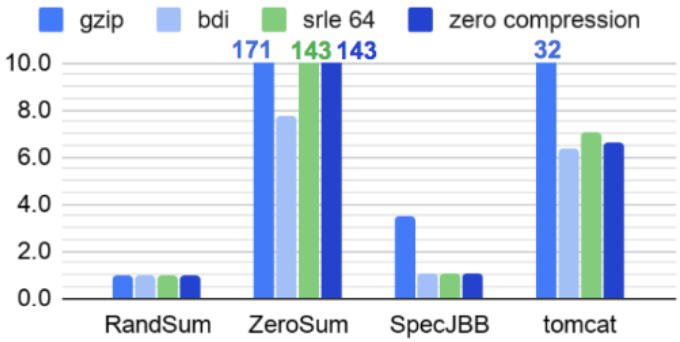

Fig. 5. Heap compressibility for various benchmarks

Fig. 5 demonstrates compressibility of various target benchmarks.

First two benchmarks are from our microbenchmark set. They demonstrate two corner cases - huge array of random numbers (not compressible) and huge array of zeros (very compressible).

Spec Jbb heap snapshot was taken at the moment when heap is almost full (almost no zeros), but even in this case we got compressibility of 3.5 with gzip compression algorithm.

Tomcat benchmarks represents another application doing transactions. It's heap is tiny (not so many objects), so, we can get good compressibility even with primitive compression techniques.

III. Interesting Observations

A. 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 element, 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 or decompression for the arrays of primitive types.

For this experiments we 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.

Fig. 6. Heap map for 1GB array summation microbenchmark

B. Empty space in arrays

In Fig. 6 red regions represent location in heap of 1 Gb array. Green regions are zero pages. We can see that a lot of empty pages exist in the heap dump. That means that heap compressibility can be overrated (because of these empty regions).

We also noticed that some of the empty regions are present in virtual memory only. No real memory was allocated in this case.

IV. Related work

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 also plan to use SpecJBB2005.

V. Conclusion

We have observed interesting overhead results in this project and learned a lot about JVM structure and optimizations. We managed to achieve acceptable overhead values in the overwhelming majority of experiments, but there are several avenues for further improvement. An abnormally high overhead we have observed in a few cases could be addressed by switching to object based or page based compression to avoid unnecessary checks. Second, we are currently investigating scalability issue for SpecJbb. Third, optimization path has a big influence on performance results.

The most interesting question raised so far is "What is compression?". Initially, we assumed that the whole heap can be compressed and our compressibility results are based on this assumption. During the research, we realized a couple things. First, only the tenured area can be compressed. Second, JVM adds empty space to the tenured area in order to prevent extra garbage collections. If we deallocate this space, garbage collection will occur immediately, because there is no space for new objects, and it might result in the application slowdown.

An important observation we make based on our experiments is that the naive ways of estimating heap compressibility are unstable, due to the evolution of the heap in time - its size, amount and size of living objects and hence compressibility can vary drastically over the lifecycle of the program. The community still needs to establish metrics to account for these changes - specifically, at which moment in time, and how exactly to measure the memory compressibility, across a variety of memory access patterns, platforms and use cases.

Our experiments show that frequent compressed memory checks can bring a significant performance degradation to an application. The only reasonable balance can be achieved for larger objects. On the contrary, pre-allocated empty memory (allocated by garbage collector) can't be compressed due to the current garbage collector policy. This brings us to the idea that only large arrays can be efficiently compressed.

VI. Future work

Table III contains the timeline of this project.

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	DONE
Get final Performance results on Raspberry Pi	DONE
Page based vs Object based compression	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

TABLE III
Project timeline

The basic evaluation of the bytecode injection based compression was done this semester. The instrumentation was tested on both server machine with OpenJ9 and Raspberry PI with Hotspot JVM.

In order to reduce instrumentation overhead, we need to make checks less frequently. One way of doing it is performing one check per object. For example, for large arrays with 100 loop iterations, the check can be done 1 time instead if 100 and it can be done outside of the array. In order to evaluate is this approach is reasonable or not, we need to find the amount of arrays in the heap dump and find its compressibility.

This will be the next step in our project. Based in the amount of the arrays in the heap dumps and the compressibility that we can get out of it, we can determine whether object based compression is reasonable or not.

Another way of improving overhead is using page-based compression. In this case compression is happening at page granularity and compressibility is potentially higher, but the same page might contain parts of different objects with different compressibilities.

One of the two ways of compression will be implemented in the garbage collector and evaluated. In the upcoming work we will try to add cold/hot objects identification and do compression based on this information.

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