Making Garbage Collection Efficient By Dynamic Offloading

Jie Tang
University of California
Riverside, California
tangjie.bit@gmail.com

Chen Liu
Clarkson University
Potsdam, New York
cliu@clarkson.edu

Jean-Luc Gaudiot
University of California
Irvine, California
gaudiot@uci.edu

Abstract—Garbage Collection (GC) is still a major issue in JVM for both mobile and cluster computing. GC offloading is proposed to improve the performance of GC by delivering part or all of the operation into another dedicated GC co-processor or accelerator. However, when offloading, the previous methods did not consider the phase change of GC behavior, which can be classified into two different groups: minor GC and major GC. The minor GC is fast and frequently invoked, while major GC is expensive in terms of time but seldom takes place. The direct offloading made GC workload frequently hopping between main processor and GC accelerator, introduced a noticeable overhead and offset any possible benefits of offloading. To solve this issue, in this paper, we propose to offload GC dynamically by a careful selection of profitable and harmful GC operations. We also made a case study on Apache Spark, a lightning-fast cluster computing platform. It shows dynamic offloading can yield nearly 42.6% performance improvement with a concurrent 32.1% in energy cost reduction.

I. INTRODUCTION

Java is widely used in embedded devices, mobile phones, enterprise servers and supercomputers[1]. However, its memory Garbage Collection (GC) is still the performance bottleneck[2]. For example, on average 20% of the CPU cycles in Apache Spark[3]. For some data-intensive applications, the overhead produced may be even higher.

As a hardware approach, GC offloading seeks to turn over a part or all of the garbage collection operations into a specialized hardware component. However, traditional approaches is direct offloading, considering no phase changes in Garbage Collection. The minor GC and major GC act significantly differently in terms of both invocation frequency and duration time. As a result, traditional ones make GC workload frequently hopping between the main processor and the GC hardware. It comes with frequent Ping-pong effects, leading to inefficiencies in both performance and energy. In this paper, we first propose a dynamic GC offloading mechanism to wisely switch GC workload from host to accelerator and give a case study on Apache Spark to demonstrate how dynamic offloading can indeed be performance and energy efficient.

II. MAJOR GC VS. MINOR GC

It has been empirically observed that the most recently created objects are also those most likely to become unreachable quickly. Based on this observation, a generational GC algorithm divides objects into generations [4]. As shown in Figure 1, the entire heap is divided into the Nursery Object Space (NOS), used to store newly created objects, and the Mature Object Space (MOS), used to store mature objects, i.e., the objects that survive one or more collections. When NOS becomes full, GC will take place in NOS to move those few live objects to MOS, so that the entire NOS region can be overwritten with fresh objects; it is referred to as minor GC. When MOS becomes full, a major GC will be triggered.

![Generational GC Algorithm](image)

III. THE PAIN OF GC FOR CLUSTER COMPUTING

We study the performance and energy efficiency of dynamic offloading in cluster systems. We experimented with Apache Spark. Spark is a fast and in-memory data analytic cluster computing framework. Our Spark cluster consists of 100 nodes, and each node is a 32-core machine from Intel. For Spark, we used an in-house benchmark in machine learning community dedicated for Area Under Curve (AUC)[5].

We sampled the GC behavior by monitoring the garbage collection in one of the nodes of a 100-node Spark cluster. We found that on average GC will take 20% of total execution time. Considering GC is pure management overhead, a 20% share is really a huge overhead. When the Spark cluster scales up, the overheads will rise sharply.

IV. WHY WE NEED DYNAMIC OFFLOADING

In Figure 2, we compare major GC with minor GC in terms of invocation frequency and duration time. All data is collected from Apache Spark and has been normalized to 1. Even though minor GC is triggered much more frequently, taking up almost
98% occupancy of total GC amount, it just consumes only 15% of entire GC time. The maximum major GC time is 127.58 ms, while even its minimal time is about 70 times longer than average time of minor GC. It’s shown that minor GC takes place at a high frequency but lasts very shortly; major GC is invoked comparatively infrequently but with a much longer execution time. It can be concluded that their behaviors in performance and energy efficiency are quite different, making dynamic GC offloading a better choice for fast and/or energy efficient GC.

In traditional offloading, once a GC is invoked, the corresponding GC operations are directly offloaded into a GC accelerator. After GC is finished, execution switches back to main processor. As shown in Figure 3, when major GC is offloaded, it can make main processor idle for a long time and into deeper sleep, so the energy cost can be consequently reduced. Inversely, minor GC makes main processor wake up right after it enters into idle mode. Sometimes the GC time is too short for the main processor to get a chance for entering idle. The direct offloading makes execution hop to co-processor for just a tiny little time and soon hop to main-processor, and then back to co-processor again. Since minor GC happens often, the hopping between main processor and GC accelerator happen constantly, acting like Ping-pong. When Ping-ponging occurs, both the main processor and accelerator can get no chance for energy/time saving; instead they must pay back the overheads for switching on/off. The more often Ping-ponging occurs, the more overhead it might introduce, and the more benefits of offloading will be offset. That is to say we should carefully calculate the net profits offloading might yield and dynamically make offloading.

In this section, we make the static analysis to show if we can choose not to offload those less profitable, even harmful GC invocations, how much efficiency we can get. We performed our validation on a case study with Apache Spark. In our experiment, we use PF for minor GC and use Mark-Sweep[13] for major GC. In our study, we evaluated on a generic GC accelerator we proposed in [6]. By examining the details of GC execution using different GC algorithms, we identified three performance hotspots in different GC algorithms: gc_alloc_fast, vtable and scan_slot, which contribute to more than 50% of the total GC execution time. By moving these hotspot functions into hardware accelerator, we can achieve an average 1.5X speedup in execution time. To estimate both the energy and time overhead for workload switch to/back, we use the energy cost of powering on accelerator in [6] and also check the user menu for power data of Intel(R) Xeon(R) CPU E5-2450 v2.

In first step, we classify all the GC invocations in the sampled period into two groups: profitable GCs and harmful GCs. For the simplicity, we just use the GC time for classification. For those profitable, we choose to offload them into the accelerator. Since it can provide 1.5X speedup, the time will be cut by half. For those harmful GCs, we just keep them running on the main processor and the time keeps the same. We sum each time up and we can get the final performance benefit brought by wisely offloading. For a cluster computing application, GC offloading can yield a 42.6% time reduction. With the power data collected in Xeon and accelerator, we can get 32.1% reduction in energy cost. The results show wise offloading GC can achieve both performance and energy efficiency.

VI. CONCLUSIONS

In hardware community, traditional GC offloading does not consider the phase changes caused by minor GC and major GC, leading to less efficient "GC acceleration". Based on this observation, we firstly proposed to dynamically offload GC for the purpose of high performance as well as energy efficiency. We also show a case study on Apache Spark that with the dynamic offloading, we can get 42.6% improvement in performance and 32.1% in energy saving separately.

REFERENCES


