Intelligent Mobile Code Offloading Decisions

Aluri, Ajit
aaluri@umich.edu

Arora, Silky
silkyar@umich.edu

Rao, Supriya
supriyar@umich.edu

Abstract

1. Introduction

2. Background and Motivation

In COMET (Code Offlaod by Migrating Execution Transparently) [**?**] the authors introduced a runtime system that allows migration of threads between systems depending on their workload. COMET extended distributed shared memory (DSM) techniques to enable offloading, a process in which a resource constrained device can execute compute intensive tasks on a relatively high performance device.

The core idea behind COMET is to manage memory consistency across different points at a field granularity using a VM Synchronization primitve. Changes to objects are tracked at a field level by setting an addtional dirty bit to true if the field was modified by an end point. The *happens-before* ordering between reads and writes of different threads helps in merging the changed fields to achieve a consistent state. A VM synchronization primitive is used to transfer the states of virtual heaps, stacks, source code (dex files) and synthetic classes to the other end point.

For every new thread that has to be migrated, a parallel thread is created on the other end point, if it does not exist already. If a parallel thread does exist, a VM sync is used to transfer state. A VM sync is usually initiated when a thread requests for ownership of an object it doesn't own or when the thread can be migrated to the other end point for execution. The thread that initiated the sync starts collecting all the fields that have been changed since last sync. Once the dirtied fields have been obtained, the stacks that have to be synced are gathered and all the state changes are then transmitted over to the other end point. During changed state collection all threads are temporarily suspended and are resumed just before the data transfer is initiated. The thread that was migrated waits in a loop waiting for a *resume* message from the other end point. Communication between parallel threads running on different end points is via such messages.

To avoid exchanging the code that the other end point has to resume from, all the source code files are transferred only on the first syncronization. The current program counter value gives the next instruction the other end point begins execution from. To support locking primitives, COMET assigns an owner to each lock (through a flag) indicating the end point that currently owns the object. When a thread encounters an object it doesn't own, an ownership transfer request is initiated to acquire the object, which could result in a VM synchronization.

Native functions are methods written in C language that are compiled to run on a specific architecture and are therefore non-offloadable. These native methods are invoked through the Java Native Interface and have a faster execution time when compared to methods written in Java. In COMET the authors identified about 200 native methods that are safe for off-loading and these are manually annotated to indicate that they can be executed at any end point. The remaining native methods are however never executed on the remote end point and are migrated back.

Scheduling decision regarding when to offload a thread primarily depends on the following factors - network conditions and time elapsed since the execution of the last native method. When a thread encounters instructions that are not to be migrated it updates the time it encountered the *unsafe point*. Migration decisions are made (from our understanding of code) periodically. The scheduler decides to migrate a thread if its execution time since a last unsafe point is higher than a threshold derived from the network conditions. Network conditions include both bandwidth and the round trip time. For the first ten syncronizations, the threshold is a weighted average between the RTT(+variance) and the recent synchronization time, after which the treshold solely depends on the average of the last 15 syncronization times.

From the implementation of COMET, we came to understand that the RTT is calcualted only at the beginning of an application's execution. We reasoned that it is infact correct to only estimate it once as the latency on evaluating it repeatedly for each scheduling decision would be atleast RTT. We observed that the last sync time (which is actually the average of the last 15 sync times) captures the available bandwidth and the network congestion. We also concluded that since last sync time implicitly captures the latency it was indeed acceptable to calculate the threshold after a certain interval, based on only the recent sync time.

We began our project with the motivation to further reduce the synchronizations between the two end points. It is clear that migrating a method that calls a native method in the future would only result in an abort at the other end point and then a transfer back to the original end point. With the aid of static analysis we wanted to predict if a method would call a native method in a fixed future interval of time. While a naive static analysis approach might immediately seem to be sufficient for our purpose, it is to be noted that by restricting the migration of a method based on its future invocation of a native method we would be forcing it to run on the host end point which could result in a longer execution time to reach the native method when compared to the same

3. Intelligent Code Offloading

4. Methodology

4.1. Call-Graph Analysis

4.2. Offloading Heuristics

5. Evaluation

5.1. Method Execution Times

5.2. Heuristics Performance

6. Related Work

7. Conclusion

References