CSCI E-98 · Final Exam



Question 1: Application

Answer 1:

VM startup time is determined to a large extent by the work done during VM initialization and the design of the Interpreter.

Initialization of the VM involves:

1. Mapping the Heap and other memory areas –like the Code Area and Static and Internal Data.
2. Setting up the Class Loader
3. Loading all the primordial classes – found in rt.jar ( standard class lib) – that are needed for any program to run
4. Initializes the other VM components – such as the JIT compiler, Interpreter, Profiler, Memory Manager etc
5. Setting up Threading System, Native Interface and the Object Model

Design Of the Interpreter: There are basically two broad approaches used to design VM interpreters

1. Simple Switch Interpreter – that parses a byte code in a big switch statement. The main feature of this approach is that the Interpreter starts running immediately, and is easy to implement. But on the flip side the Interpreter is very slow which may impact performance overall unless optimized or run along side compiler of some sort.
2. Template Based Interpreter – In this approach the Interpreter is Custom built by choosing a template for each bytecode. A linker program then links all these templates together to form the Interpreter. The main feature of this approach is that it is more efficient as the templates are optimized for each platform. On the other hand this approach frontloads the system, which adds to the Startup time of the VM.

Minimize Early Performance Cost:

In order to minimize the startup time of the VM and hence the early performance cost, I will first look at the Interpreter implementation of the VM. There could be a couple of option, which can be investigated to study the start up times

1. If the current set up of the VM uses a Template Based approach, trade this with the simpler version, compile that so it runs closer to that machine. We could balance this with a combination of Profiler + JIT Compiler. With this approach the startup time can be reduced without hugely swinging the overall runtime of the system
2. If the management could clearly identify the top 5-10 targeted platforms for the VM, we could provide a mix of Template based Interpreter pre build/configured, for those top platforms and fall back to the simpler approach for the rest. Since the Template based Interpreter will be pre-build for the targeted platforms, the startup time will not pay the time penalty for building it on demand.

Question 2: Memory Management

Answer 2:

Requirements of Memory Manager:

Following summarizes the requirements of the new memory manager and their impact to the overall design of the Manager.

1. Application software contains significant native code implemented using JNI:

The presence of native component in the application software plays a significant role in the overall design, specifically because native runtime environment is managed separately from the VM. Native Code is unaware of the VM data structures, and book keeping concepts such as safepoints, write/read barriers, stack maps, which the VM relies on to manage its runtime and provide accurate GC. There are now two different runtimes and objects allocated in each and referring each other as needed.

Objects in the Native environment make use of direct access to pointers, and any GC algorithm needs to carefully consider how they manage references from native environment to managed environment avoiding the situation of dangling pointers. Local and Global references from the Native code are managed separately. Local References are created as a result of either passing references as parameter to the method or returned as value from a method call, and are mostly managed automatically. These references are no longer needed once the call is finished. A reference needs to be promoted to a Global state by explicitly setting it. These references are managed by the Native code.

1. Hardware setup of the application
   1. Single core processor with 2 hardware threads:

Since there is only a single core and 2 hardware threads, we need to understand what process the VM currently runs in parallel. If one of the threads is set up to run profiling and JIT complication in parallel with application run, it limits the ability to do GC in parallel.

* 1. Small main memory:

This impacts the size the Heap that will be available to run the application. And in turn effects the allocation Strategy, in that we may not be able to have a portion of heap that is not used ( like in Semi space Copying Collector) and need to be mindful of heap fragmentation.

* 1. Word addressed memory – with 32-bit word size:

Having a memory addressed by word boundary can play to our advantage, if the allocation can be done aligned by a word boundary. This gives the ability to go right to the address, instead of having to find the offset to go to, for fields. We could use technique of bit stealing to store metadata and save on the already limited memory available. Smaller metadata and smaller object footprint is advantageous as it leads to fewer GC cycles as less heap fragmentation. Large objects also impact cache, which is again somewhat limited on the current platform.

* 1. Caching:

Caching is usually is an important factor in the design of memory manager in general, but the long latency between L1 cache and RAM on this platform makes this even more critical. If objects can be allocated word aligned and are such that can fit in one cache line (which in the case of L1 D-cache is 64 bytes per line), then by accessing any part of the object we can get the whole object into the cache. So having the header next to object can be very useful strategy as accessing the header will bring the whole object into cache removing the performance overhead of round trip to main memory.

* 1. Ability to run VM in embedded systems:

Embedded systems usually have fewer core and smaller memory, very similar to the current platform. As the company plans to sell this VM for use in embedded systems, GC cannot have very long pause times.

GC algorithms: Options and Trade-offs

This section discusses the algorithms studied in the class for coming up with the design of the Allocator and Garbage Collector for the Memory Manager of the VM.

1. Allocators:

Object Model and Use of Handle Vs Header: The design of the Object model has an impact on the allocation algorithm. As the application has limited memory and we would like to take advantage of aligning objects by word boundary, using the approach of an Object Header, which sits close to other object data can be very useful. On the other hand using Handles as a way to store object metadata, can be useful specifically because the application has native code, which can access managed objects and needs to always have the right reference to the object.

The two Allocation algorithms are

1. *Bump Pointer Allocation*:

This algorithm has a simple allocation strategy, where the allocation pointer starts at the start of the heap and moves forward by the size of the object that is allocated until it reaches the end of the heap. This algorithm does not deal with Heap fragmentation and so cannot be used if the references pointed too by the native code are Pinned. If we were to use Handles as a way to get to the references we could probably use this algorithm, but would need to update reference in two locations

1. *Non Contiguous Allocation*:

This algorithm handles heap fragmentation that may result from object pinning strategy, as it reuses any available free space to do allocation – based on algorithm chosen – such as Best Fit, First Fit and Worst Fit.

1. Garbage Collectors:

The algorithms for Garbage Collector are

1. Reference Counting:

This algorithm tracks references that are made to an object and collects the object when the count goes down to zero. The reference count is usually stored as a part of the metadata for that object i.e. in the Object Header to the Handle, using the technique of bit stealing.

Some advantageous of this algorithm is that it has low pause times and the object gets collected as soon as it is identified as garbage. Some of downside with reference counting are, update that may need to be reflected across multiple objects, which may in turn hurt cache and reference overflow. Reference counting algorithm will need an allocation strategy that deals with fragmentation.

1. Copying Collectors:

In this technique the heap is divided into regions and allocation is made to one region called the Nursery and objects that survive GC, get promoted to another region called the Mature space. This GC technique uses Bump Pointer Allocation for object allocation. When the Nursery gets filled GC is triggered. Object graph is traced starting with root set to identify all the live objects. These live objects are copied to the mature space and the allocation starts in back in Nursery.

In a semi space copying collector, the heap is divided into 2 halves and regions are flipped after each GC.

One advantage of the Copying collector is that there is no heap fragmentation, which make allocation easy and also fast. But the GC pause time can increase linearly as the size of the nursery increases. This is due to the fact that, in order to do GC the collector usually stops the world and scans the whole nursery to identify live objects. There is also the cost of copying objects and update reference.

1. Mark Sweep Collector:

Another form of Tracing collector, but works by identify objects that are garbage. There is no copying of objects, objects are allocated as needed and get collected when they become unreachable. There are 2 phases in this collector (A) Mark Phase – which identifies and marks the live objects, starting from roots. (B) Sweep phase – where all unreachable space is reclaimed and made available for the next allocation. Allocation is made using the Non Contiguous allocation technique.

This technique has not copying overhead and has pause times characteristics similar to Copying Collector. On the flip side, it does lend itself to heap fragmentation.

1. Generational Collectors:

Use a mix of Copying and Mark Sweep Collector, with Nursery using the Copying collector with Bump Pointer Allocation and the Mature space being managed by Mark Sweep with Non Contiguous Allocation.

The aim is to keep Nursery allocation and collection fast and manage the copying overhead of objects that have survived multiple GCs. Overall the strategy aims to maximize throughput. But the implementation is more involved and can get complicated specially when trying to do these concurrently.

1. Running these collectors in parallel:

Any of the above collectors can be run in parallel with separate threads, but the correct implementation of this can be challenging and should only be undertaken when there is hardware that can support the parallel run.

Proposed Design:

A proposed design for the VM that can meet the current requirement would be using a Mark Sweep Collector and allocation done using Non Contiguous Allocation Strategy. The metadata for the Object will be in the Object Header. Object Header itself will be 2 Word size, where first word is reserved for Class Descriptor and other word will be used as Mark Word – that can have information of Mark bit, useful for GC. Native object references will be managed using Object Pinning.

Question 3: Refactoring

Answer a)

1. Refactor the code so that it interacts with as many classes as possible early in the application’s run, forcing the class loader to load all classes early:

The rationale for this refactoring is that, dynamic class loading makes optimizations much harder. Each time a class is loaded dynamically, it kicks off a Class Hierarchy Analysis across all loaded classes. Class Hierarchy Analyses inspects the set of classes loaded by class loader to determine what optimizations are valid. It then rolls back any optimization by were made invalid by the newly loaded class. By front loading all the classes the application will not pay the penalty the comes with de-optimizating.

This optimization is useful in that it reduces the de-optimization cost of an invalid optimization caused by Dynamic Class Loading, and provides a optimized code for the application to run. But Dynamic Class Loading is a useful and powerful feature, which provides flexibility such as adding new types/classes.

1. Minimize the use of object orientation, in order to maximize the number of static method calls:

The reasoning for this refactoring is that InvokeVirtual byte code is much more expensive than InvokeStatic. InvokeStatic byte code is used when a Static method is called and InvokeVirtual is used by the instance methods. Invoke Static is efficient as the exact method is coded into the instruction, while Invoke Virtual triggers a Class Hierarchy Analysis to determine the dynamic type on which the method needs to be executed.

For methods that do not store any state and are like functions – that take an input and return a value – this optimization can apply. But if the method changes/mutates the state of the object, it cannot be defined as a class level (or a static method).

1. Re-implement parts of the code in a native language and integrate with the VM using JNI:

Native code runs faster to any code that is Interpreted and/or Compiled, as it runs, that much more closer to the machine. Native code is already optimized by the native compiler.

Although native does run faster, integrating it with JNI presents other sets of challenges to the VM in terms of GC, and maintenance of this code.

1. Implement an object pooling strategy that pre-allocates and re-uses objects rather than creating new instances on demand.

This idea is trying to reduce the GC and Object initialization cost. Object allocation can trigger GC at any point based on the state of the Heap. Additionally based on the type of the GC algorithm being used the same objects could be copied over and over in every GC cycle.

This strategy can be targeted to a finite set of objects, where the initialization cost of the object is considerable (for example database connection), and objects that are needed for the lifetime of the application. For any other objects this option can lend itself to memory leak and more GC.

1. Reduce the size of synchronized sections, and replace explicit monitor acquisitions with atomic operations.

Atomic (CAS) operations are coded into hardware so does perform well, but comes with its overhead of thread synchronization at the hardware level. Most objects are usually not locked and locks are reentrant. Contention between threads for lock, is expected to be low/rare.

We need to study the code to understand the execution of synchronized section of the code. If there is a fair chance of contention between threads, replacing with CAS atomic operation is a better option.

Question b) :

Answer b) : Some profiling information that can be useful

* + - 1. Timing and Logging information – Is needed to identify what part of the application code needs to be tuned for performance
      2. GC – The current GC profile and configuration of the application, to understand how objects are being allocated and get collected, and those that are long lived.
      3. Threading – Identify what portions of the application need to be concurrent.
      4. Hardware - The Hardware on which the application runs
      5. Object Graph - The type hierarchy, information of the polymorphic call sites, information around objects that are needed for the lifetime of the application.

Question 4: Highly Concurrent JVM

You want to optimize the VM for concurrent throughput, even at the expense of single-thread performance. You must implement the JVM specification (i.e. you cannot change the memory model, and your system must implement priority scheduling). For each of the following components, discuss any design decisions that you might make to emphasize multi-threaded performance:

(a) The implementation of monitors, including the contention manager that determines which thread should obtain a monitor or be scheduled from a set of waiting threads

Answer (a): Monitor or Locks are primitives that make synchronization between threads work. Synchronization can be of two types

1. Mutual Exclusion: where only one thread can enter the critical section of the code
2. Cooperation: where multiple threads coordinate and message each other to complete some work.

Biased Locking:

In this case, the requirement is to optimize throughput even at the expense of a single –thread, a Biased Locking Strategy can be useful. Biased locking is designed with an assumption that Locks are reentrant. If some information – such as Thread Id, - can be stored in the Object/Monitor to indicate that it is Biased towards that threadId, then that thread can acquire the monitor without using an atomic CAS operation.

Contention Manager:

Since locks are mostly reentrant this can be a useful optimization. In cases where there is true Contention, and thread trying to acquire the lock is not the one for this monitor is biased, the monitor will be re-biased to the new thread(Id).

In cases, where the application continuously uses different threads to execute critical section, this approach can prove to be expensive.

Answer (b):

Per JVM specification, any object can be a monitor and monitors are reentrant, meaning the same thread can acquire the monitor multiple times.

To do this, some information is needed per object to

1. Indicate whether or not the object is locked.
2. If the object is Locked
   1. which thread holds the lock and how many times has it acquired the lock. Since locks are reentrant, having this information allows the currently holding threads to reacquire the lock.
   2. lock needs to be released as many times as it was acquired by a thread. Additionally Lock needs to be released by a thread so other threads can acquire it.

Object Model: The Object Header of heap allocated objects will be a 3 word header,

* The first word holds the information of the class descriptor
* The second word will hold an optional array length
* The last word will be used to indicate the Lock state of the object. It will store the ThreadId, of the thread that currently holds the lock. The last 2 bits of which will be mangled to represent the count of number of times it is locked by this thread. When the Object is not locked this word will be empty

Answer (c):

The scheduling mechanism: Priority Queue

As we need to implement Priority scheduling the Entry Set – which represents that threads trying to acquire the lock – can be implemented as a priority queue.

Each thread object will have a priority number and the priority queue will order them based on this number. If more than one thread have the same priority, the order in which the thread entered the queue will be used to determine which of them will get scheduled. The Wait Set, can be a implemented as a HashSet and any thread that goes into the wait state will get added to this Set. On receiving a notification signal ( using notify() or notfiyAll() ), the waiting thread(s) will be moved to the Entry Set, which maintains the queue based on the priority.

The scheduling Manager calls the remove() method on this queue of Entry Set, to schedule the next thread

Question 5: Defeating Decompilation (100 points)

Answer 5: To identify what can be changed in the generated byte code, I took a look at the bytecode and .class files of classes in the sample\_code that we have been using

Some of the post processing steps I could think of

1. Rename classes to something more cryptic, (like java compiler creates a made up name for anonymous classes) so they make less sense if they get decompiled
2. Rename variables/Methods: We can rename variables and methods to be less meaningful. This one I really wanted to tryout. So I first looked at the javap output of StaticFieldAccess and then opened the class file in vim.

I changed the variable name in class file from intVar to i.tNar (using hex editing in vim) and saved the file and ran the program. The output was unchanged, meaning the code semantics were left intact, but when I did a javap on the new class file , output changed from

Good one :

#6 = Utf8 intVar

#8 = Utf8 intVar2

BadOne :

#6 = Utf8 i

tNar

#8 = Utf8 intVar2

I am not sure if the decompiler will trip on it, but most certainly the translated source code will be hard to understand. We can make this even more difficult by renaming to something really cryptic.

1. Add methods/variables that are not really used anywhere in the code :

So I added a method with native access modifier in StaticFieldAccess.java. I did this directly in java code – new method :public native void testMe();, but I am sure we can probably add this directly into the byte code. Since we know that the JIT compiler will optimize this and remove any unused methods, this should be ok. But perhaps this will confuse the reader of the source code. The javap output after I added the native method, had the below lines, so decompiler will also have them, making the source code slightly more tricky.

public native void testMe();

flags: ACC\_PUBLIC, ACC\_NATIVE

1. Remove the debug information: Removing information such as LineNumber- from the byte code , will make the generated code from the decompiler harder to follow and will not have close resemblance to source code.

(5) Some techniques used in JIT compilation: Some of the techniques we discussed in class for JIT compilation - as inline, loop unroll, Constant folding – changed the original source, and was at sometime harder to understand compared to the original code. Some of these techniques can perhaps be used to post process the bytecode, and will definitely make the decompiled code very different to the source code.