malloc() & free() in Scoped Memory

Adding Manual Memory Management to the Fiji™ VM

Checkpoint II

Prepared For:

CSE 605 - Advanced Concepts in Programming Languages
Dr. Lukasz Ziarek, Department of Computer Science & Engineering
University at Buffalo, The State University of New York

Prepared By:

Janhavi Patil (jpatil@buffalo.edu)
Mihir Mehta (mihirmeh@buffalo.edu)
Shreyas Chavan (schavan2@buffalo.edu)
Scott Florentino (scottflo@buffalo.edu)

24 April 2015

Abstract

High level programming languages have many attractive features but lack predictability. Fiji solves this problem by converting Java bytecode to C. The Real-Time Specification for Java (RTSJ) , developed as JSR under the Java Community Process, defines additional interfaces and specifications that extend Java to a real time context[1] . One of these interfaces is the concept of Scoped Memory. The RTSJ defines scoped memory where a piece of memory is associated with a particular scope and allocated like a runtime stack. In scoped memory, objects can be allocated at runtime and once allocated, cannot be deallocated. The entire scoped memory area is deallocated when there are not any references to it. ScopedMemory allows NoHeapRealTimeThreads to run without the threat of GC stopping the world.

Adding malloc() and free() to Scoped Memory allows the developer of a NoHeapRealTimeThread to still perform memory allocation when necessary and free memory when possible. Since the objects allocated by malloc() are still released when the thread leaves the scope, we reduce (but do not eliminate) the chance of a memory leak (although the leak would be bounded to the size of the scope only). In addition, while malloc() and free() place additional burden on the developer, we argue that developer memory management within a particular scope is much more manageable than management throughout an entire codebase. In addition, malloc() and free() do not *replace* the new() (or the "Java" way) of allocating objects in scoped memory. We propose that both methods can co-exist side-by-side. Similarly, a RealTimeThread is free to allocate memory on the Heap and rely on the Garbage Collector. Our proposal allows for developer optimization of real-time code - and its completely optional, at that.

The next section will be a brief revision of our proposed implementation, followed by the modifications that we had to make to FijiVM™ and the challenges we faced. We will further explain in detail how we've implemented malloc() and free() in FijiVM ScopedMemory followed by some tests that we conducted. The later section will deal with what could be added to our implementation.

Recap of proposed Implementation and Tracking structure

We have proposed fragmented memory allocation where each chunk is of size 64 bytes. These chunks are used to store primitives and arrays of primitives. 7 primitives can be stored in every chunk. Similar to Schism, arrays are managed by using a sentinel and array spine approach. The sentinel contains the metadata size and information about the array spine. The array spine contains the pointers to the individual array elements.

Our tracking structure was fairly simple and consisted sets for all free blocks, blocks with some free space and fully filled blocks. We have implemented these sets with a linked list whose head is popped for allocation. In array allocation, the spine will not be allocated if the entire array can reside in the same chunk as the header. Optionally the spine can reside in

the same block. If neither of these cases occur, the header contains the pointer to the array spine.

In our implementation we have asked the user for the amount of memory in the scoped memory reserved for malloc() and free() and allocated a free chunked linked list in this memory. And as allocations happen we have casted these fivmr_nodes to either fivmr_primitive_blocks or fivmr_array_blocks. Pointers to headers of these lists along with a lock are included in fivmr_MemoryAreas_s.

Changes implemented in Fiji

As we had proposed in Checkpoint 1 we have made subsequent changes in fivmr_MemoryAreas_s and added structures for fivmr_node, fivmr_primitive_block, fivmr_array_header and fivmr_array_block. We have included wrapper classes for hooking up Java and C in fivm/runtimej/src/com/fiji/fivm/r1/unmanaged along with UMUtils that contains the native calls to the C methods fivmr_MemoryArea_allocatePrimitive for allocation of a primitive and fivmr_MemoryArea_deallocatePrimitive for deallocation of a primitive. There is one such wrapper class for each primitive type but the methods call the same native methods. These classes have get(), set(), allocate() and deallocate() methods. An illustration of UMInteger wrapper class is given below.

```
12  public class UMInteger implements UMPrimitive {
13      public static int get(Pointer p) { return p.loadInt(); }
17
18      public static void set(Pointer p, int val) { p.store(val); }
22
23      public static void free(Pointer primitive)
24      {
25          //TODO
26          final Pointer curArea = MemoryAreas.getCurrentArea();
27          if(curArea == MemoryAreas.getHeapArea() || curArea == MemoryAreas.getImmortalArea())
28          {
29              throw new UnsupportedOperationException("Deallocation cannot occur in Heap or Immortal Memory");
30          }
31          //Call to native to Deallocate
32          fivmr_MemoryArea_deallocatePrimitive(curArea,primitive);
33      }
34
35      public static Pointer allocate(int val)
36      {
37          //We can't be in the Heap OR Immortal Memory
38          final Pointer curArea = MemoryAreas.getCurrentArea();
39          if(curArea == MemoryAreas.getHeapArea() || curArea == MemoryAreas.getImmortalArea())
40          {
41              throw new UnsupportedOperationException("Allocation cannot occur in Heap or Immortal Memory");
42          }
43          Pointer primitivePointer =  fivmr_MemoryArea_allocatePrimitive(curArea);
44          primitivePointer.store(val);
45          return primitivePointer;
46      }
47  }
```

Figure 1: UMInteger Wrapper Class

Once the user enters the Scoped Memory area, he can allocate an Integer by calling UMInteger.allocate() method with the integer as a parameter. This method will in turn call the

native method to get the location where the primitive is to be stored. The native method store() in the Pointer class is then used to store the primitive at the aforementioned location. The method returns a pointer to the location of the primitive. We can get the value of the previously allocated primitive with the UMInteger.get() method by passing the pointer to the integer. We can also mutate the value of the integer with the UMInteger.set() method which takes the pointer to the integer and the new value as parameters. The programmer can deallocate the primitive by calling the UMInteger.deallocate() method which in turn calls the native method to deallocate the primitive whose location is passed as the parameter. The same procedure applies to all the other primitives.

Primitive allocation logic is as follows: We determine if the fr_head list is empty - if so, remove the head of the free block list (O(1)) and make it a fivmr_primitive_block (O(1)). Allocate the primitive in the block, updating its bit vector ((O(1)). If the block becomes full after allocating the primitive, remove it from the list of half-filled blocks and add it to the list of fully-filled blocks (nfr_head). If there is a space for allocating the primitive in the linked list of half-filled blocks, allocate the primitive in this linked list according to first-fit strategy.

Primitive Deallocation logic is as follows: We determine the start of the primitive block (subtract start from the primitive address - and floor the result to the nearest 64 byte offset from the start) then find the bit vector and update it appropriately. This has the effect of a O(1) operation in most cases. However, if the designated block which just had the primitive removed is removed from the nfr_list and added to the fr_list, there is a higher penalty. We faced a challenge here which was moving a block from nfr_list to fr_list. We were successful in adding this block to the fr_list but we tried to remove it from the nfr_list, we realized that we had to traverse the nfr_list to see which block was to be removed. Since the list is singly-linked, we must traverse the entire list to well as remove it. We had not initially considered this additional traversal time, with is O(n), with n being the size of the unmanaged area. While we haven't benchmarked this yet, we believe it will have minimal impact in practice because this happens most often with a large amount of primitives being in use on the system. We argue that an array should be used at that point. In addition, we can easily modify the allocation mechanism to prefer to leave fr_blocks almost (but not completely) fulfilling them only when absolutely necessary. This further reduces the event in which a primitive block must be moved from the nfr_list to the fr_list. Again, this O(n) penalty only occurs when a primitive is freed from a data block which was full - deallocations from data blocks that are only partially full only incur an O(1) penalty, since no lists need be traversed.

Array Allocation and Deallocation

Array allocation and deallocation was implemented as proposed (with one exception) in Checkpoint 1. While the original plan was to use Java array syntax, we should that this would require significant modifications to the Fiji Runtime - introducing additional checks and possible race conditions. Standard array behavior differs in Fiji depending on the scope in which is it executing in, as well as the array's size (a small size may result in stack allocation). However, the goal of this project was to change the array behavior *without* modifying the Scoping structure in Fiji. As this is only checkpoint 2, we decided to move forward with the implementation and correctness testing *first* rather than implementing Java-compliant syntax.

While Java-compliant syntax is a must for any released product - it is not needed for testing and benchmarking, as it is trivial to run benchmarks with differing array semantics. We do realize, however, it is *not* trivial to write standard programs with differing array semantics. We will address this later in the report.

That being said, there is two differences in the array implementation from our initial proposal. Instead of using a C enum to enforce type checking, we use a simple integer in the array header. Because normal Java array semantics were dropped, we provided a type-safe Java interface for array allocation, element access, and element mutation. We use a Java enum that maps to a C integer, and that is compared in the header. The performance impact of this has yet to be determined - for the Fiji runtime provides type-safe methods for primitive arrays already. However, if we adapt to Java semantics, we can avoid this Java-side runtime type checking. In addition, we keep track of the current amount of memory allocated in unmanaged memory in the fivmr_MemoryArea C structure. This allows us to to report the memory usage to the user as we provide a function similar to MemoryAreas.consumed() [in fact, it is called MemoryAreas.consumedUnmanaged()], which provides the developer with the current amount of memory allocated in the unmanaged area. We also noticed several cases in which our design document from Checkpoint 1 had C structures of incorrect sizes. We corrected the appropriate padding to create 64-byte C structures in order to align with our 64-byte block size.

Our original proposal did not detail the actual memory layout of ScopedMemory once we had implemented our design. While we originally planned to go with allocating memory at the end of the scope, slowly creeping towards the start of the scope (just as the stack and heap approach each other in an address space), we actually allocate the scope of the specified size at the beginning of scoped memory, and push the start of scoped memory “out” of the unmanaged area. This simplified development and ensured we did not break other parts of the runtime unintentionally - we followed the bump pointer semantic to allocate an area for ourselves within the Scope. While this does result in a section of scoped memory automatically being reserved for unmanaged memory, this also has the effect of greatly simplifying the layout of unmanaged memory; we are guaranteed to have a constant number of blocks and can easily detect and out of memory error in O(1) time. If were dynamically changing the boundaries between scoped and unmanaged memory, this increases the time complexity required to allocate unmanaged memory, since all block addresses must be checked to make sure they weren’t claimed by scoped memory.

The interface to unmanaged arrays is as follows:

Here, we provide 4 methods - allocate/free (essentially acting as malloc & free here), and accessors/mutators. In addition (but not shown in the screenshot), there is a length() method; given a Pointer to an array, will return its length. These operations are type-safe Java syntax, but do assume that a Pointer object actually refers to an Unmanaged array. It would be difficult, if not impossible, to perform proper checking & safeguards as demanded by Java in this manner - for a Pointer object could quite literally point anywhere. Once arrays are integrated into the Fiji runtime and use Java array semantics, this should be a non-issue.

```
public static Pointer allocate(UMArrayType type, int size)
{...}

public static void free(Pointer array)
{...}

public static int getInt(Pointer array, int index)
{...}

public static void setInt(Pointer array, int index, int val)
{...}
```

We also utilize the same array checking semantics that the Fiji runtime uses. For example, consider access for an integer array:

```
public static int getInt(Pointer array, int index)
{
    nullCheckAndArrayBoundsCheck(array, index);
    //Type check
    if(Magic.semanticallyUnlikely(CType.getInt(array,"fivmr_um_array_header","type") != 0)) {
        throw new ClassCastException();
    }
    return fivmr_MemoryArea_loadArrayInt(array, index);
}
```

While we do add our own type checking here, we utilize the nullCheckAndArrayBoundsCheck from the runtime. In addition, we pass the value to C via a native method to be inserted into the array.

Additional Implementation Notes

In addition to implementing our proposed design, we also created a unified memory interface for both primitive and array allocation - since both involve the same block size. This allows a single point for detecting and handling out of memory errors - namely, throwing an OutOfMemoryError in Java. While no formal test exists for this yet, we observed this particular feature working properly many times while we were debugging and benchmarking our implementation.

We also began adding some C macros for idioms such as block size, elements per block, and so one. We would like to further expand on this for our final implementation, as it would allow for easy modification of parameters such as block size if additional performance tuning requires it.

We found we had to edit some VM assertions as the the offsets for some runtime data structures had changed due to our additions to the fivmr_MemoryArea type. We also integrated capacity checking with the GC interface in the runtime. Most importantly, our implementation does not penalize the performance of developers who choose not to use it; so long as they are content with an extra 16 byte size of the fivmr_MemoryArea C structure (which, in theory, could also be eliminated with a duplicate struct without our metadata).

We also formally defined the overhead we incur when allocating an array in the unmanaged area. For areas with sufficiently small payloads (<7 elements), we inline the payload with the header, meaning no spine is needed. We take this inspiration from Fiji itself. We could also add a pointer to a single data block to store additional elements, if the size is still relatively small, as is also done in Schism, but we have not done this yet. Currently, all primitives are given 8 bytes in arrays, meaning there is some storage overhead when storing data types that are not 8 bytes large. This is a trivial implementation that can be taken later. That all being said, the overhead for arrays larger than 6 is 12.5% the size of the array - that is, the spine that points to all datablocks in memory. It is calculated by using the following (note that all elements are 8 bytes, and the block size is currently 64 bytes):

Math.ceiling(elemCount / ELEMENTS_PER_BLOCK) * sizeof(void*))

This equation is used extensively in our tests to calculate the proper sizes for scoped and unmanaged memory, as we tested on very tight bounds to ensure absolutely no memory was leaked.

Remaining Implementation Details and Future Scope

The previous sections cover the preliminary implementation of malloc() and free() in scoped memory in Fiji. Both implementations function as designed, and are tested to the point where we can verify that are not leaking memory under the following conditions:

- Single thread allocation/deallocation
- Single scope / unmanaged memory region (no nesting)

While many error conditions are accounted for (such as null checks, bounds checks, and out of memory errors), there are a few edge cases that need to be accounted for and tested:

- Allocating an unmanaged memory segment of negative size
- Allocating an unmanaged memory segment too large for the underlying scope
- Allocating an array with negative size
- Allocating an array too large for the current state of the system
- Properly throwing an OutOfMemoryError and other Throwables in where appropriate (for simplicity, fivmr_asserts were used in some parts of the C code during development)

In addition to accounting for these conditions (better to raise a Throwable than abort the VM), we plan on adding tests for these conditions to ensure they are handled properly. Given the current testing framework, this should be trivial. In addition, unmanaged arrays for data types other than ints have yet to be implemented. However, this is trivial and doesn't require any design changes. However, the following changes might:

- Supporting synchronization of multiple threads using an unmanaged area of a scope.
- Verification that the scope nesting rules are properly enforced
- Finding a way to further mitigate or completely avoid memory unclaimable overhead by breaking up the spine into chunks while still providing O(1) random element access for arrays.
- Supporting Java array semantics
- Supporting better unmanaged primitive semantics

- Supporting object allocation/deallocation

Initial Correctness Testing

As discussed previously, both primitive and array allocation were tested rigorously to ensure they functioned as designed, did not leak memory, and did not disrupt operation of the Fiji VM. Below is a screenshot of one of the array correctness tests:

```
for(int i =0;i<6;i++)
{
    //Perform allocation in random order
    Collections.shuffle(activeArrays, r);
    for(int j =0;j<activeArrayCount;j++)
    {
        int arrayIndex = activeArrays.get(j);
        arrayPointers[arrayIndex] = UMArray.allocate(UMArray.UMArrayType.INT, elemCount);
        Pointer array = arrayPointers[arrayIndex];
        for(int k=0;k<elemCount;k++)
        {
            UMArray.setInt(array, k, values[arrayIndex][k]);
        }
    }
    //Perform verification in random order
    Collections.shuffle(activeArrays, r);
    for(int j =0;j<activeArrayCount;j++)
    {
        int arrayIndex = activeArrays.get(j);

        Pointer array = arrayPointers[arrayIndex];
        for(int k=0;k<elemCount;k++)
        {
            int cur = UMArray.getInt(array, k);
            reportError(cur ==  values[arrayIndex][k], 3); //Value misplaced!
        }
    }

    //Perform deallocation in random order
    Collections.shuffle(activeArrays, r);
    for(int k =0;k<activeArrayCount;k++)
    {
        int arrayIndex = activeArrays.get(k);
        UMArray.free(arrayPointers[arrayIndex]);
    }

}
```

In this test, several arrays (in this case, 6), are (1) allocated, (2) populated with random values, and (3) deallocated. Note that we perform each of these operations in random order. Some data structure allocations are not shown - in this particular test - we calculated *exactly* what this operation would need and provided on that amount of memory. This means that some structures need to be allocated outside of the scope, marked as final, and access where appropriate (the random values were generated outside of the scope). Thus, we can only print out primitives to the console since we technically don't have the memory to create a String buffer. While this may have been overkill for our initial tests, we wanted to make sure we could handle maximum memory constraints. When implementing tests that check for

invalid conditions such as negative sizes, we plan on using large enough scopes to allow for printing of error messages inside the scope instead of just integers. Even with just the error codes, the tests structured in such a way that the user is still notified of a failing test. The reportError() routine actually makes calls on the heap which print “ERROR: {error code}“ clearly indicating to the user that a test has a problem. In addition, we print the start and finish event of every test, also in Heap memory, to provide information on currently executing tests. In each test catches a Throwable, so if an error occurs that shouldn’t (or should, for that matter) that is also verified and reported to the user.

```
reportError(MemoryAreas.consumed(MemoryAreas.getCurrentArea()) == overhead + unManagedSize , 1); //"Scoped Memory was leaked!"
reportError(MemoryAreas.consumedUnmanaged(MemoryAreas.getCurrentArea()) == 0L, 2);// "Unmanaged Memory was leaked!"
```

Initial Benchmarking Testing

While additional stress testing and benchmarks against standard Heap operations (which support reclamation, like unmanaged memory) still need to be developed and performed, we have a simple but extendable set of tests that have allowed us to begin reasoning about the performance and space requirements of our implementation. Initially, we chose to stress test our implementation by perform matrix multiplication with a highly fragmented memory region. This was implemented twice: (1) on the Heap using the Fiji VM and (2) in Scoped Memory using our unmanaged memory interface, also on the Fiji VM.
Our test computer had the following specifications:

AMD Phenom II X4 945 Processor (4 x 3.0GHz cores)
16GB DDR3-SDRAM @ 1333MHz
Ubuntu Linux 14.04 (64-bit)
32-bit test binary

Our test had the following specifications:

1) Fragment the heap by generating 100 randomly sizes arrays from size 1 to size 10,000
2) Create two 1D arrays of size 10,000 (simulating two 100x100 matrices), and assign random values to each element
3) Create another 1D array of size 10,000 (simulating the resultant 100x100 matrix)
4) Perform matrix multiplication on the two input arrays (simulating two 100x100 matrices), assigning results to the resultant matrix

We use 1D arrays because these were all that is currently available from our implementation. We recorded the time, in nanoseconds, required to perform each of the operations (2) through (4).

The test above was repeated 10 times in the same VM, and the figures were averaged, once for each type of operation.

The Fiji VM used to test CMR Heap memory had the following memory limit to ensure fragmentation and pressure on the GC: 140K (was an extra 20K beyond what was required by the computation)
The Fiji VM used to test HF Heap memory had the following memory limit to ensure fragmentation and pressure on the GC: 2x140K = 240K (was an extra 140K+2x20K beyond what was required by the computation). With only 140K of space, the test ran out of memory.

The Fiji VM used to unmanaged memory did not have a memory restriction, as it was set by limiting scoped memory. *However*, The size (effectively a limit) on scoped/unmanaged memory was an extra 20K beyond what was required by the computation (the unmanaged memory region size was 240192 bytes + the 20K extra space). Note that the enclosing scoped memory was much larger (2M) due to the overhead of allocating a large number of arrays.

The results are as follows:

	Heap (CMR)	Heap (HF)	Unmanaged Memory
Array Allocation	150,656 ns	370,068 ns	38,900 ns
Array Element Mutation	7,003,764 ns	6,252,790 ns	7,786,323 ns
Matrix Multiplication	62,008,351 ns	100,253,641 ns	370, 922,406 ns

While unmanaged memory allocation is significantly faster at allocation than a CMR heap, its lower mutation throughput hindered its ability to perform matrix multiplication, resulting in an order of magnitude slowdown. *However,* the CMR test frequently had to be re-ran because it did not tolerate fragmentation and the VM crashed. This was not the case with unmanaged memory, which is fragmentation tolerant.

The Schism/CMR garbage collector (HF) also outperformed unmanaged memory during computation (but not allocation), although it required more memory than either [we are not counting the overhead of the pre-computation fragmentations on the unmanaged memory scope, which was roughly 2M].

Our results were surprising, as the unmanaged memory scheme did not have any complexity due to fragmentation; instead incurring a space penalty. However, our implementation is preliminary and has not been optimized; in addition, we are performing a type check on every array access. Ideally some of these checks could be enforced (at least for primitives) at compile-time, as they are for standard arrays. The large amount of static function calls we make in our implementation due to our circumvention of Java array semantics may be to blame as well.

Future testing plans include:

- More trials than just 10 (in practice, we ran several groups of 10 to verify the consistency of test results).
- Larger matrices
- Varying amounts of fragmentation & free space
- Benchmarking of array element access (we already have mutation)
- Additional computations, ideally as real-world as possible, that could determine the effectiveness of our implementation.

Conclusion

Scoped memory defined and implemented in Fiji did not previously have the dynamic allocation and deallocation. Objects once allocated in Scoped Memory could not be deallocated. This lead to memory being wasted during the execution of computations in a scope if objects or arrays were allocated. We have tried to overcome this limitation by introducing malloc() and free() in Scoped Memory. We have implemented a solution by defining the necessary structures to track objects which have been allocated by the ‘new’ operator as well as our implementation of malloc(). Our preliminary implementation includes allocation and deallocation of primitives and arrays of primitives.