malloc() & free() in Scoped Memory

Adding Manual Memory Management to the Fiji™ VM

Final Report

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

Checkpoint 1 Recap [full content available in Checkpoint 1 PDF]

The RTSJ defines scoped memory where a piece of memory is associated with a particular scope and allocated like a runtime stack to add predictability to Java. 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. malloc() and free() do not replace the new() of allocating objects in scoped memory. We propose that both methods can co-exist side-by-side.

We have proposed fragmented memory allocation to provide 100% fragmentation tolerant allocation and avoid need for cleanup. These chunks are used to store primitives and arrays of primitives. For array allocation we used a strategy similar to Schism GC viz. using array sentinels and spines with pointers to the individual elements. The only difference was that in our implementation the spine would reside in the Scoped Memory.

Our tracking structure were sets implemented by linked lists for each type of block.
We left it to the programmer for the amount of memory in the scoped memory reserved for malloc() and free() however, we argue that developer memory management within a particular scope is much more manageable than management throughout an entire codebase.

Checkpoint 2 Recap [full content available in Checkpoint 2 PDF]

In Checkpoint 2, we have made 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.

Primitive allocation and deallocation

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.

Array allocation and deallocation

Array allocation and deallocation was implemented as proposed except that we dropped the idea of having the Java array syntax as it would require significant modifications to the Fiji Runtime - introducing additional checks and possible race conditions. 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 also used a simple integer to enforce type checking. For array interface, we provide 4 methods - allocate/free (essentially acting as malloc & free here), and accessors/mutators. In addition, there is a length() method; given a Pointer to an array, will return its length.

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 ensured that 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.

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. We also began adding some C macros for idioms such as block size, elements per block, and so one. We edited 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 did not penalize the performance of developers who chose not to use it; so long as they were content with an extra 16 byte size of the fivmr_MemoryArea C structure.

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.

Introduction

This section of the report builds on the on the previous work done in Checkpoints 1 & 2. While we began benchmarking in previous checkpoints, our primary goal of this checkpoint is to complete verification of both correctness and performance. In addition, we provide additional insight to the tests we performed. We discuss some additional implementation issues, and describe our framework for testing.

Benchmarking Overview

Our benchmarking suite was extended for the final report. Previously, we were only multiplying a 100x100 matrix and measuring the wall clock time it took to complete this computation. Many other variables were not discussed. We increased the workload of the benchmark significantly and added additional benchmarks. Below is a table describing our improvements:

Metric	Checkpoint 2	Final Project
Array Length (square of matrix size)	10,000	160,000
Fragmentation Passes	100	1000
Max Fragment Size	33%	66%
Memory Usage	200K	3M
Trial Count	10	25
Comparison to JVM	No	Yes
Cache/TLB Analysis	No	Yes

By adding additional stress factors to our implementation, we hoped to get consistent results. We did this by porting the benchmarking code to standard Java, and profiling on the JRockit JVM on the aforementioned system. While this is not an exact 1:1 comparison, since the JVM operates very differently than Fiji, we used the profiler to get an idea of the memory layout of the operations we were performing.

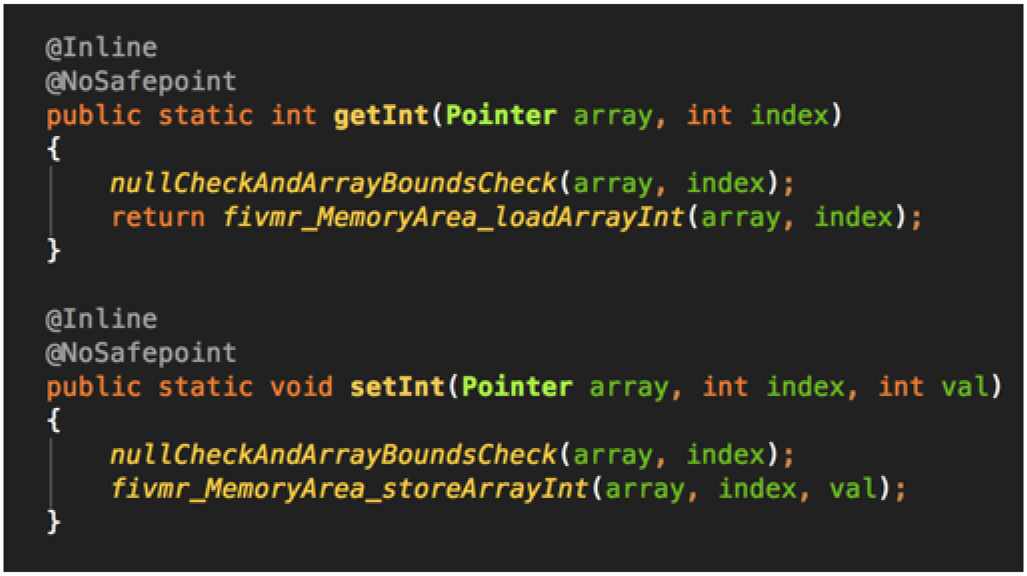
Previous Performance Bottlenecks

As described in Checkpoint 2, we noted a significant performance disparity between our implementation of malloc() and free() in scoped memory and current GC models. Obviously this is not desirable. While many sacrifices to performance are made in the name of predictability, our implementation was essentially an order of magnitude slower than two implementations that already provide predictability. Here is a brief recap of the results:

	Heap (CMR)	Heap (HF)	Unmanaged Memory
Array Allocation	150,656 ns	418,748 ns	38,900 ns
Array Element Mutation	7,003,764 ns	6,294,357 ns	7,786,323 ns

Matrix Multiplication	62,008,351 ns	98,474,551 ns	370, 922,406 ns

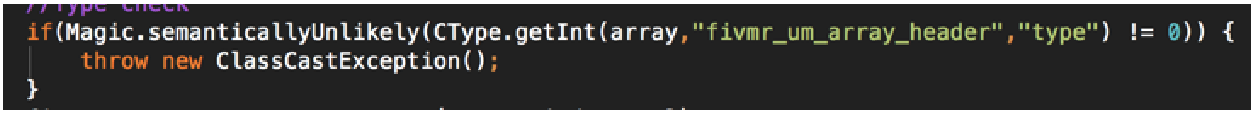
Since we are extending features of the Fiji VM without modifying the Java syntax or compiler, much of our functionality had to be implemented with method calls (the Java 8 feature of virtual extension methods would have been great here).



```
@Inline
@NoSafepoint
public static int getInt(Pointer array, int index)
{
    nullCheckAndArrayBoundsCheck(array, index);
    return fivmr_MemoryArea_loadArrayInt(array, index);
}

@Inline
@NoSafepoint
public static void setInt(Pointer array, int index, int val)
{
    nullCheckAndArrayBoundsCheck(array, index);
    fivmr_MemoryArea_storeArrayInt(array, index, val);
}
```

Each of these methods has functionality to verify valid indexes and perform null reference checks. These methods are type safe only in their second two arguments; not in their first argument. As this is an experimental research prototype, we assume that the caller is passing a valid pointer to an array they malloc'd using out interfaces. However, in Checkpoint 2, we did attempt to enforce type safety with array, and performed the following check on *each* array access and mutation:



```
//Type check
if(Magic.semanticallyUnlikely(CType.getInt(array,"fivmr_um_array_header","type") != 0)) {
    throw new ClassCastException();
}
```

However, as this is a prototype implementation, we decided to forego runtime type checking of primitive arrays, as with proper compiler support we can provide compile-time type checking. Using Java array syntax would require significant architectural modifications of the Fjii VM, as we are sidestepping the Memory Area paradigm and changing the allocation behavior *inside* a particular Memory Area. One possible way to do this *without* syntax modification is via a flag of some sort before and after the array allocation, but this is vulnerable to race conditions. As both Java syntax modification and extensive Fiji VM modifications do not provide extensive benefits for actually prototyping our implementation, we opted not to perform them.

In addition, several small optimizations were made to the UnManaged array access and allocation code, to improve performance. For one, null checks for array access were performed using Pointer.zero(), which was not an inline function. Offsets calculation was simplified. Some debugging statements were also removed.

Current Performance Metrics

With the removal of runtime type checking, several CPU instructions were removed from each individual array element access, thereby reducing the already-O(1) associated cost. In addition to this modification, we extended the benchmarking suite as already noted to perform our tests.

The first benchmark we performed was on primitives.

Below is the outline of the test carried out for measuring the timing information of Primitive allocation/deallocation:

- We allocated a 1000 integers, both on the heap and in the scope.
- We used the CMR and the HF GCs for the heap and used our implementation of malloc() and free() for allocating/deallocating integers in the scope.

The results were as follows:

	Heap (CMR)	Heap (HF)	Unmanaged Memory
Integer Creation	59,447 ns	15,933 ns	62,461 ns
Integer Access	3456 ns	3140 ns	2140 ns
Total	425,380 ns	351,447 ns	876,881 ns

The next benchmark we performed was matrix multiplication, requiring about 2MB of memory for storage of matrices *a*, *b*, and the resultant matrix, *c*. Both the CMR and HF trails were given the same amount of heap space (1.4x the size of the memory required to perform the matrix multiplication itself), as the overhead required for HF's semispace allocation converges to roughly 1.5x. Our unmanaged implementation, being 100% fragmentation tolerant, was given a fixed size overhead of 150K. Below are the rough steps of the test (timed portions are denoted below)

-Calculate and initialize Memory Areas, if needed

-Fragment the heap with 1000 passes, with each pass generating an array of a random size from 0-160,000

-Allocate *a* and *b* (Array Allocation)

-Populate *a* and *b* with random values (Array Element Access)

-Verify *a* and *b*'s values (Array Element Mutation)

-Perform matrix multiplication (Matrix Multiplication)

This operation was performed for 25 trials and averaged on the following system:

Intel Core i7 3610-QM (4 physical cores, 8 virtual cores) @ 2.3GHz-3.3GHz
8GB DDR3-1333 SDRAM
SATAIII SSD

Ubuntu 14.04 x64

The results of the computation follow:

	Heap (CMR)	Heap (HF)	Unmanaged Memory	JVM (64-bit JRockit 7)
Array Allocation	537,608 ns	736,555 ns	382,308 ns	4,760,240 ns
Array Element Access	296,589 ns	676,489 ns	1,183,968 ns	196,622 ns
Array Element Mutation	8,665,398 ns	6,382,481 ns	6,456,606 ns	8,985,866 ns
Matrix Multiplication	182 ms	585 ms	1026 ms	107 ms

We can draw several interesting conclusions from this data:

Array Allocation

Array allocation is significantly faster with unmanaged memory than any other implementation. This is to be expected; as our implementation is 100% fragmentation tolerant and all of the measured operations are operating with extreme heap fragmentation (to simulate the worst possible case). With unmanaged memory, we simple grab blocks in O(arraysize) time and reserve them. This is a fairly inexpensive operation and doesn't involve barriers or on-demand fragmentation.

Array Element Access

This is where our implementation falls short. While it uses many of the Fiji runtime constructs to perform null checking and array bounds, it still falls short of current implementations. This is likely due to several reasons. While we inline null checking and array bounds operations, using the current Fiji runtime array optimizations that are available to us, we don't have bytecode-level support and accessing array elements still requires calling a static function and the runtime level, before even hitting runtime code. With compiler and bytecode-level support, likely several optimizations could be made to this routine. In addition, unmanaged arrays are not continuous, requiring 2 memory accesses instead of one (first, the spine in accessed, then the data block the spine points to). A check for arrays of inlined size is also performed, resulting in a branch. The JVM and CMR implementations do not suffer from this bottleneck.

Array Element Mutation

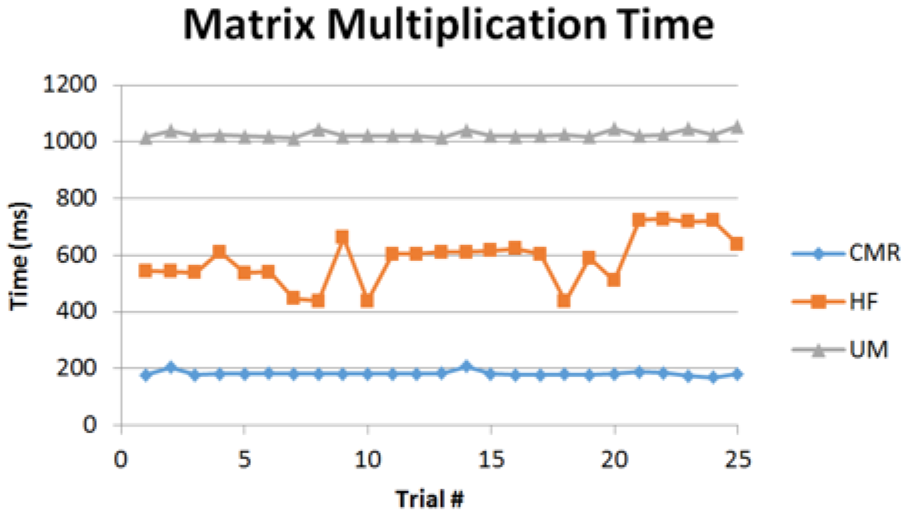
Surprisingly, our implementation fares quite well here; we do perform two memory access instead of 1, as we again have to access the spine and the data block. However, as we are fragmentation tolerant and have no read or write barriers, our mutation code is fairly simplistic.

Matrix Multiplication

Matrix multiplication involves a combination of the above operations, and the metric is expected. As it involves more reads than writes, the slowdown in array access dominates the runtime for our implementation.

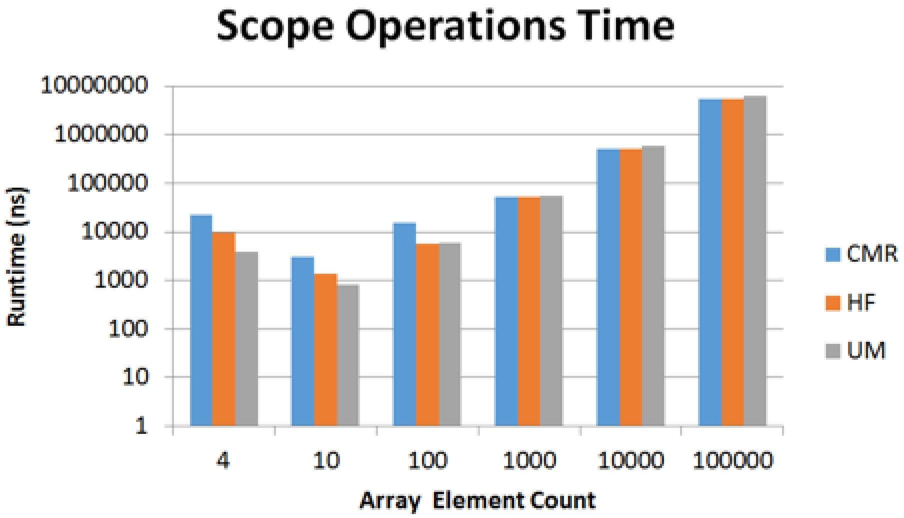
Predictability

In addition to performance, we also analyzed the predictability of our implementation. The complexity of each operation is discussed in Checkpoint 2, but as previously stated is fragmentation tolerant. We graphed the runtime of each operation to see the trends of predictability. Despite the increase in overall runtime, unmanaged memory still has small variations in runtime. Note again that the longer time is due to increased time for array element accesses.



In addition to matrix multiplication, we took additional metrics to attempt to capitalize on reduce allocation time, and compare our overhead to the allocation/deallocation of scopes, as that is essentially a sidestep to GC activity just as malloc() and free() would be. In addition, this could be a potential source for speedup compared to CMR/HF, and could potentially cancel out the penalty for decreased array access times. For this benchmark, we shifted the focus from the array operations themselves to the cost of allocation/deallocation/entering/exiting scopes. For the scoped-based approach, a new MemoryArea was allocated and entered into for each array that was created (we create 500 arrays, assigning values randomly to each in random order). For the unmanaged-based approach, we use a single scope and allocate/deallocate 500 arrays inside, performing the same operation. This test was perform for several different array sizes, and the results were recorded and graphed.

Array element sizes 4-1000 are shown on the graph; the entire results are shown in the table.



Array Element Size	CMR Runtime (ns)	HF Runtime (ns)	UM Runtime (ns)
4	22270	9585	3846
10	3053	1386	821
100	15021	5652	5844
1000	51433	51298	56299
10000	505473	506719	593639
100000	5541985	5526914	6346032

For small array sizes (particularly inlined ones), unmanaged memory significantly outperforms both Fiji GC implementations when large amounts of scopes are used. Of course, as array size increases, the penalty for array accesses in our implementation drowns out any gains encountered from avoiding the creation of nested scopes. Interesting is the delay for inlined arrays (size = 4); such a size is often unlikely in practice; hence predictions are probably missed with sizes this small. If array access penalties could be reduced, environments with a high rate of scope allocations/deallocations could benefit even more from our implementation.

Correctness Tests

Several correctness tests were implemented in Checkpoint 2 and verified in the final stages of development, to ensure proper implementation. These included:
-Inlined, Small & large array allocations, where values and memory usage were verified
-Interleaved unmanaged and scoped allocations, where values and memory usage were verified
-Creation of scopes with unmanaged memory, where memory usage was verified
-Creation of scopes with invalid arguments, were exceptions were verified
-Deliberate creation of out of memory errors, where exceptions were verified

These tests are bundled with the source code, and ensure memory is not corrupted *and* the proper errors are thrown when needed.

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 leading to wastage of memory during the execution of computations in a scope if objects or arrays were allocated. By introducing malloc() and free() in Scoped Memory we have addressed this limitation. We 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. We carried out a few tests to verify the correctness and check the performance of our implementation and documented the results.