VMKit - Adaptive Optimization System by using LLVM JIT Compiler

Gayana Ranganatha Chandrasekara Pilana Withanage,Kuganesan Srijeyanthan, David Daharewa Gureya
{gayana.withanage, kuganesan.srijeyanthan, david.gureya } @tecnico.ulisboa.pt
Group 2
Instituto Superior Técnico, Lisbon, Portugal.
May 16th 2014

Abstract

This article is presented to propose an improvement for an existing problem regarding the performance of LLVM JIT (Just in Time) compiler, which is an inherent slowness during the start-up of virtual machines (VM), Ex: VMKit J3 [3]. Our proposed solution is to use a JIT compiler as an adaptive optimization for the current VMKit implementation. What's left is a system that can keep track of and dynamically look-up the hotness of methods and re-compile with more expensive optimizations as the methods are executed over and over. This should improve program start-up time and execution time and will bring great benefits to all ported languages that intend to use LLVM JIT as one of the execution methods. Moreover our implementation will have a mix mode execution including a non-optimized and dynamic optimization with LLVM JIT. In the end, we will benchmark our implementation so that we can compare the difference in performance of the current LLVM JIT and the improved LLVM JIT with VMKit framework.

***Key words**: VMKit, LLVM, JIT, MMTK-Garbage collection*

1. Overview

Implementing a virtual machine (VM) is a painful task which demands huge effort and knowledge regarding the hosting infrastructure and architecture. When moving from one architecture to the other it is again a time consuming task which demands reimplementation of the main modules of VM. To relax this inherent overhead of building a VM, VMKit which work as a substrate has been developed so that it provides about 95 percent of the code required in developing a new VM. VMKit provides basic components required to create a VM such as JIT (Just In Time) compiler, Garbage Collector (GC) and a Thread Manager.
J3 and N3 are VMs that have been developed using VMKit which prove the significant reduction in development time of a completely new VM [3]. Furthermore, the core of VMKit depends on LLVM compiler infrastructure which provides the required key components such as JIT compiler and GC. In this article our point of interest is the LLVM JIT compiler because it is a known fact that the startup time consumption during the VM execution, developed using this core module is considerably higher. Therefore we propose an improvement for the behavior of LLVM JIT compiler, in order to make the startup and the run time to be faster than the current implementation.

2. Virtualization technology studied

- ***Name***: VMKit
- ***URL***: http://vmkit.llvm.org/
- ***VM-type***: High Level Language VM
- ***Common usage***: VMKit eases the development of new managed runtime environments (MREs) and the process of experimenting with new mechanisms inside MREs. J3(JVM) and N3(CLI) are built on top of VMKit.
- ***Motivation***: While VMKit aims to provide a fully functional intermediate layer for VM, its still under heavy development and misses some vital components with some open projects proposed. One of the current drawbacks of the LLVM JIT is the lack of an adaptive compilation system which will be our contribution to this project. What is missing is a system that can keep track of and dynamically look-up the hotness of methods and re-compile with more expensive optimizations as the methods are executed over and over. This should improve program startup time and execution time and will bring great benefits to overall performance.

3. VMKit

VMKit is a common substrate that eases the development of new managed runtime environments (MREs) and the process of experimenting with new mechanisms inside MREs. It provides the following to the end users: a precise garbage collection, Just-In-Time and Ahead-of-Time compilation and portability on many architectures. For the MRE developer, VMKit provides: research and development infrastructure for virtual machines and a relatively small code base. The main goal of the VMKit is to help experiments on virtual machines.

VMKit uses LLVM as the JIT compiler (generates the native code on the fly), MMTk as the memory manager (allocating and collecting free memory automatically) and POSIX Threads as the thread manager (creating and synchronizing threads).

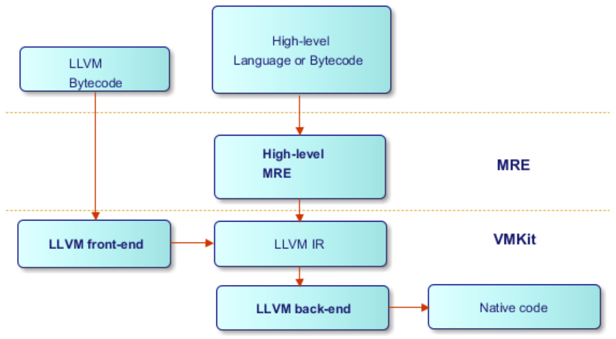


Figure 1: Compilation process in VMKit

3.1 The JIT Compiler

The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. All the libraries including VMKit have been implemented in C++. As VMKit is a substrate for VMs, the components integrated in order to develop VMKit should have a general purpose interfaces and functionalities. Therefore the compiler should have support for a general purpose instruction set to allow VMs to implement intermediate representations of arbitrary functions. LLVM JIT is exactly a suitable compiler for this task because of its inherent behaviour without imposing object model or type system.
Once the VM generates the intermediate representation (IR) with the help of interfaces provided by LLVM, that VM delegates the compilation to LLVM JIT to generate the native code. LLVM JIT optimizes all methods to the same degree without considering the frequency of the number of time each method get called. This behaviour is called the compilation without adaptive optimization. As a result, the initialization time get higher and the runtime also shows some slowness.
The LLVM IR is made up of five instructions as follows

- Arithmetic, copy and cast operations on registers
- Local control flow instructions (branch)
- Method invocation (direct and indirect)
- Memory reads and writes
- Intrinsics

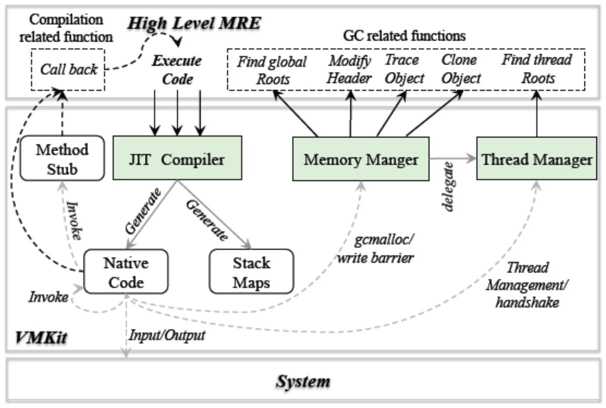


Figure 2: VMKit architecture

3.2 Compiler extension with intrinsics

Any language should be able to map to the LLVM IR. Since LLVM directly targets the C language, there are many instructions that are not extensible in C that could be used by MREs, e.g. atomic instructions. LLVM supports this instructions by means of intrinsics: an intrinsic is a contract between the language and the compiler to generate a specific assembly instruction. It is materialized as a function call in LLVM. Beside atomic instructions there are other processor instructions, like vector operations, math operations and stack operations that LLVM implement as intrinsics.

3.3 Lazy Compilation

The execution engine performs lazy compilation of the methods throughout the lifetime of the application i.e. in case a method calls another method which has not been translated or compiled yet, the execution engine inserts a callback. When called, the callback invokes the high-level MRE method to locate the new method.The high-level MRE provides the callback function. It loads and generates the intermediate representation of the lazily compiled function, and finally delegates to LLVM the generation of its native representation. Once the function is generated, LLVM patches the calling site to call the newly generated function for subsequent calls.

4. Memory Management

VMKit uses the Memory Management Tool Kit (MMTK) which is a well-designed and optimized toolkit for writing high performance memory managers. MMTK responsible for memory allocation and reclamation. Here in this section we discuss mainly the composition of MMTK and its functionality as a Garbage Collector (GC) and later we explain how it has been integrated with the VMKit.

4.1 What is MMTK?

MMTK is an efficient and extensible memory management utility developed using Java [2]. Because of the selection of a high level language like Java, MMTK has obtained the additional feature of portability which makes it flexible, moreover the clear definition of the interface has enabled it to integrate with different VM with lack of overhead.

4.2 Composition of MMTK

Mechanisms, Policies and Plans are the key compositional elements of the MMTK architecture. The virtual address space is divided typically into 4MB chunks and those are managed with the help of policies defined specifically. In a specific Policy the memory management Mechanism has been implemented. Plans are the highest level of composition which are used to generate Policy objects in order to manage the heap. Figure 3 below illustrates class diagram of the MMTK policies while Figure 4 shows the class diagram of MMTK plans.

Following standard allocation and collection mechanisms are implemented by MMTK [1]

- ***Bump Point Allocator*** - When it is requested to allocate memory for a certain object, that object will be appended to the end of the free memory and bump pointer will be incremented to the size of that object.

- ***Free-List Allocator*** - Memory is organized into a segregated free list which keeps a linked list of free memory blocks of a predefined size. When a new object arrives allocation point can be identified from the list with lower search overhead.

- ***Tracing Collector*** - After identifying the live objects by stack maps and write barriers, it reclaim memory by copying them out of the space or by freeing untracked objects.

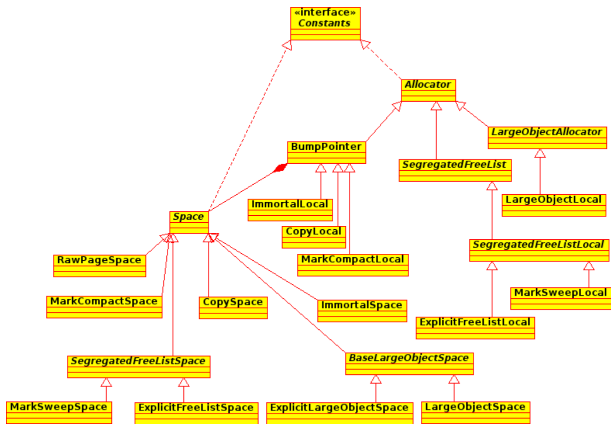


Figure 3: MMTK policies

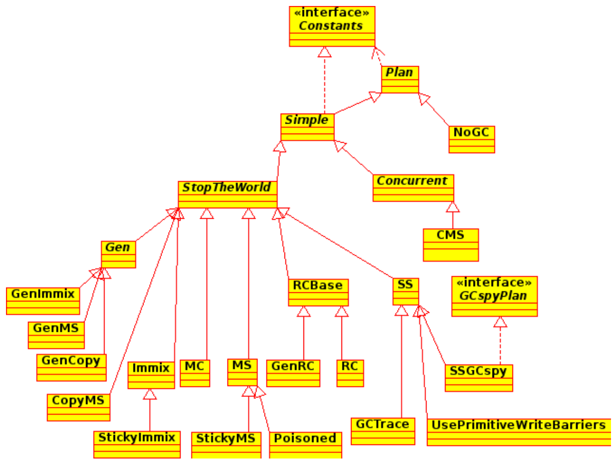


Figure 4: Class diagram of MMTK plans

- ***Reference Counting Collector***- Keep a count of incoming references for a certain object. If the count become zero, reclaim the memory.

Combining the functionalities of above mentioned basic mechanisms, **Policies** are implemented. Below is the description of few policies.

1. Copy Space uses the bump pointer allocation and tracing to trace live objects and copy it to a different location in the memory.

2. MarkSweep policy uses the free list allocation and tracing collector to track the live objects and reclaim the dead object to the free list.

3. RefCount policy, here the reference counting collector will claim the non-referenced memory to the free list.

Plan is to instantiate several policies in order to manage the memory. As an instance,

1. SemiSpace uses two copies of Copy Space. Memory is divided into two segments, one segment will be allocated while the other is free. When this allocating segment is filled live objects move to the other free segment, and vice versa at next round.

2. GenCopy, the intuition is to keep one Copy Space for the young objects and transfer the long living objects to an SemiSpace. When the Copy Space is filled then it commences transferring surviving objects freeing the memory, moreover if the old space is filled, entire heap will be collected.

4.3 Virtual Machine Interfacing

Due to the flexibility feature of MMTK it is essential to make the interface of MMTK to the Virtual Machine much clear and well defined. The interface needs to be bidirectional across two entities so that each entity should satisfy specific requirements and features.
Memory Manager demands requirements such as identification of sources of pointers, low level memory operations such as memcpy, hardware timers, atomic memory operations, I/O etc and virtual machine should cater these requirements as a feature set of the VM. VM pushes the root objects which are the global and local variables on the threads' stacks and registers to MMTK's queue and MMTK enumerates the object pointers and proceed collection.
Meanwhile the VM demands requirements such as allocation, write barrier implementation and general statics (Heap size, GC count) from the memory manager. Having this mutual cohesion, VM and MMTK can be integrated. Now in the below sub sections we discuss how VMKit has interfaced with MMTK.

4.4 MMTK integration in VMKit

According to the discussion in Section 3, the VMKit is a substrate for developing Managed Runtime Environments(MRE). Therefore the requirement of the Memory Management module is the independence of other object models or type systems. MMTK has those uniqueness by its design, therefore it is a wise selection for the memory manager in VMKit.

Even though it is a better selection, below two significant issues have to be rectified in order to integrate MMTK to the VMKit.

1. Linking MMTK with VMKit, as MMTK is implemented in Java while LLVM and the rest of VMKit are implemented in C++.

2. Providing MMTK's GCs with exact knowledge of the object location in the heap and the threads execution stacks.

First issue has been resolved with the help of J3 which is a JVM built on top of VMKit. According to the figure 5, J3 is used in order to load and compile MMTK during the compilation of VMKit. At this point the J3 is executed with a lightweight memory manager and rest of the VMKit uses the LLVM ahead of time compiler to generate the LLVM bytecode of MMTK. After generating the LLVM byte-codes those are not dependent on other runtime environments, now the VMKit loads and dynamically compiles them before launch a high level MRE.

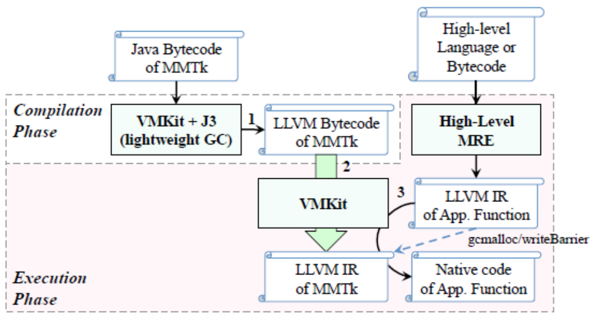


Figure 5: Integration of MMTk in VMKit. Solid arrows are compilations or transformations. Dashed arrows are invocations. The big solid arrow is the initialisation phase of VMKit

As a solution for the second issue, the MRE has to provide the functions for the GC to accurately scan the heap. These functions are highly dependent on the execution stack layouts and the object model of the MRE and those are indicated in figure 2.(See the box GC related functions.)

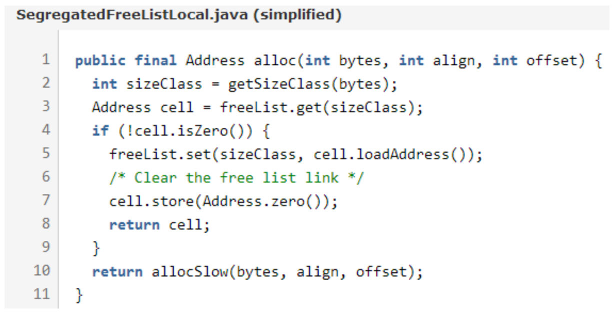
Moreover, the GC does not have the knowledge of the object layout as well as the global roots of object graph which depicts the global variables, constant objects, classes, fields etc. Therefore the MRE needs to provide those required details with the help of LLVM built in intrinsic. For instance , the intrinsic called gcroot which is provided by LLVM takes an abstract register as an argument which contains a reference and generates the stack map (Stack map indicates where the root objects are located in a frame). Furthermore MRE provides functionality for the MMTK to trace the objects so that GC can traverse the directly reachable sub objects of each object.

Once the high level MRE is loaded the mutual interactions between the interfaces of VM and MMTK takes care of the memory management of the system as a whole.

4.5 Allocation

MMTK provides two methods for allocating memory. One is the unsynchronized thread local allocation which is used by each policy by means of thread local classes while the other one is slower synchronized slow path allocation which is related to the global memory abstraction shared by multiple mutator threads. The idea here is allocating memory from thread local buffer and when that memory is fully allocated additional memory will be allocated from the global space.

For instance below code segment is a simplified version of allocation of a segregated free list implementation which initially checks the thread local buffer and if unsuccessful move for the slow allocation method which will ultimately uses a global policy.

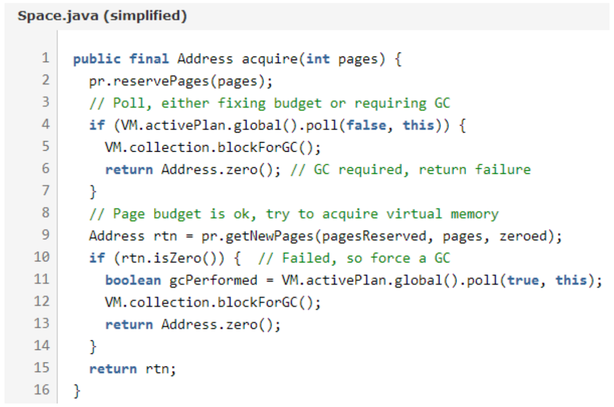


```
SegregatedFreeListLocal.java (simplified)

public final Address alloc(int bytes, int align, int offset) {
  int sizeClass = getSizeClass(bytes);
  Address cell = freeList.get(sizeClass);
  if (!cell.isZero()) {
    freeList.set(sizeClass, cell.loadAddress());
    /* Clear the free list link */
    cell.store(Address.zero());
    return cell;
  }
  return allocSlow(bytes, align, offset);
}
```

Figure 6: Segregated free list implementation

Eventually all the policies will acquire virtual memory by calling the acquire method of the Space global object. Below code segment shows a simplified view of that method.



```
Space.java (simplified)

public final Address acquire(int pages) {
  pr.reservePages(pages);
  // Poll, either fixing budget or requiring GC
  if (VM.activePlan.global().poll(false, this)) {
    VM.collection.blockForGC();
    return Address.zero(); // GC required, return failure
  }
  // Page budget is ok, try to acquire virtual memory
  Address rtn = pr.getNewPages(pagesReserved, pages, zeroed);
  if (rtn.isZero()) {  // Failed, so force a GC
    boolean gcPerformed = VM.activePlan.global().poll(true, this);
    VM.collection.blockForGC();
    return Address.zero();
  }
  return rtn;
}
```

Figure 7: Simplified view of the space global object implementation

Here it contacts the plan which know the policy objects generated for memory management to check whether the heap is fully allocated, if so wait for the GC to collect and free the memory, otherwise return the requested page's memory address.

4.6 Collection

Memory reclamation can be scheduled in two ways as stop the world garbage collection and parallel garbage collection. In the former schedule, mutator (application) are executed until the memory is exhausted and then the mutator threads are suspended until the garbage collection is completed. In the latter schedule collector threads are allowed to execute parallel with the mutator threads.

As mentioned in the previous memory allocation related details, in each attempt to access fresh virtual memory force to initiate a garbage collection. Every garbage collection is a flow of multiple steps.

5. Architecture of the proposed system

The following diagram depicts the simplified version of J3 JVM with some of the important components which have been added and modified to perform our goal which is an adaptive approach during an execution of the program. Currently J3 JVM including VMKit doesn't provide any adaptive mode of execution which is basically switching to different level of optimized code dynamically during a runtime. Current implementation of the J3 directly absorbs user's Java byte code and converted into in intermediate form called LLVM IR and finally it is generated into machine code according to the platform which the program being executed. Once LLVM IR has emitted into machine code VMKit doesn't have a direct control over the instruction being executed in the back-end architecture . Instead of Jitting all the functions into machine code VMKit only compiles whenever functions are being invoked very first time. So, subsequent invocation of the compiled function will be fetched from memory rather than coming back to front end every time of the function calls.

It would be more efficient if we could able to collect runtime information of each functions which are being executed in back-end and replace the most frequent function with the new level of optimization. In order to achieve this goal and to demonstrate the adaptive approach in VMKit the following components have been added into the existing VMKit implementation.

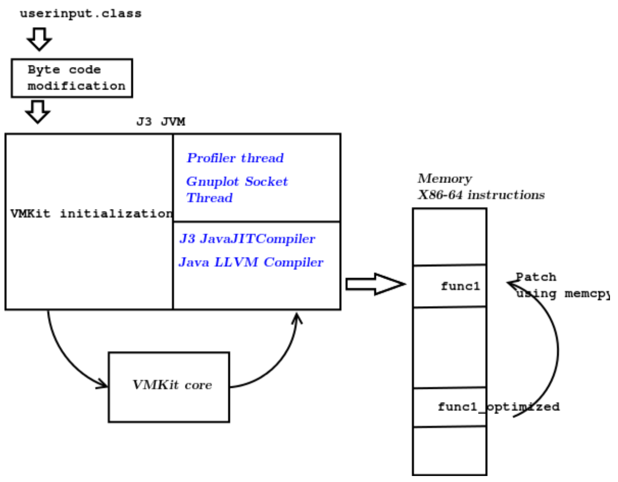


Figure 8: Modified components in existing VMKit

5.1 Byte code modification

One of the basic information which has to be collected to perform function call frequency is obviously a number of time that particular function has been invoked. This will be achieved by hooking some byte code information in the given user's Java byte code. Every function in the Java byte code will be modified with different unique global counters and extra functions also will be added to return the corresponding global counters. The following Java snippet shows how given class file will be modified.

```
 public class Source
{
   public void calculateAnswer(int input)
     {
       int  result = input * 5;
     }
    public static void main(String[] args)
      {
          Source source = new Source();
          source.calculateAnswer(4);
      }

}
```

Above Java source file will be converted into following format before we actually pass to J3 JVM

```
public class Source
{
   private static int group2S_SourcecalculateAnswer=0;
   private static int group2E_SourcecalculateAnswer=0;
   public void calculateAnswer(int input)
     {
       ++group2S_SourcecalculateAnswer;
       int  result = input * 5;
       ++group2E_SourcecalculateAnswer;

     }
   int group2S_SourcecalculateAnswer()
     {
      return group2S_SourcecalculateAnswer;
     }

   int group2E_SourcecalculateAnswer()
     {
      return group2E_SourcecalculateAnswer;
     }

    public static void main(String[] args)
      {
          Source source = new Source();
          source.group2S_SourcecalculateAnswer();
          source.group2E_SourcecalculateAnswer();
          source.calculateAnswer(4);
      }


}
```

Even though this modification will be taking place in the byte code level, actual Java source file and corresponding modified source has been given here for reader's convenient. The above actual source file has been modified by adding two global variable for each function which will to be used to achieve the following goals.

1. Need to precisely identify whether the particular function is still spending time on a specific function block
2. These two counters will be checked during the replacement of different level optimization

Further, global variable return functions also will be added as shown in the modified Java file to let the J3 profiler thread to access the global counters in the specific interval. These global functions have forcefully been called by main method so that J3 knows where these global variables are being placed in the memory.

5.2 LLVM-IR Optimization

Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program. Our implementation uses two different passes which are non optimized passes and full optimized passes. So, in the initial start up, all the functions will use the non optimized passes to generate the LLVM-IR and eventually fully optimized passes will also apply to generate more optimized code. The following example function with some unused code shows the non optimized LLVM IR. Without any optimization passes huge LLVM IR will be generated as a result. After all the passes, most of the redundant code blocks have been eliminated and as a result smaller machine code will be emitted.

```
public void Calc1()
    {
      ++group2S_GameCalc1
      double x=0.001;
      for (int z = 0; z < 10000; z++)
           {
              double a=2+x;
              double b=x+a;
              ouble c=x+b;
              double d=x+c;
              double e=x+d;
              x+=e;
            }
      ++group2E_GameCalc1;
    }
```

Unoptimized LLVM IR of Calc1 function

```
; Function Attrs: noinline nounwind
define void @32(%JavaObject*) #6 gc "vmkit" {
   ---------------
   ---------------

"false IF_ICMPGE": ; preds = %afterSafePoint4
  store double 2.000000e+00, double* %stack_double_0
  store i32 0, i32* %stack_int_1
  %33 = load double* %double_1
  store double %33, double* %stack_double_2
  store i32 0, i32* %stack_int_3
  ----------------
  %84 = load i32* %stack_int_1
  %85 = load double* %stack_double_0
  store double %85, double* %double_1
  %86 = load i32* %int_3
  %87 = add i32 %86, 1
  store i32 %87, i32* %int_3
  br label %13
}
```

The above IF_ICMPGE will be completely removed during the optimization passes.

5.3 Profiler thread

Once J3 JVM started up, Profiler thread also will be fired to keep evaluate the global counters of each functions which have been already been compiled into machine code. Each and every function will be concatenated with responsible class name to generate a key and value as a C structure which will be having the following fields. This profiler thread is fully responsible for monitoring function statistics , generating necessary information to calculate function call frequency and finally patching the optimized version of high frequency function.

```
typedef struct Patch {
        void *llvmGlobalVariableStartMachinePointer;
        void *llvmOptimzedFunctionPointer;
        void *llvmGlobalVariableEndPointer;
        Function *llvmFunctionPointer;
        JavaMethod *meth;
        void *codePointer;
        size_t Size;
        Class *cusotmizedFor;
        unsigned long long previousexetime;
        int numberofTime;
        double avgexetime;
        int previousglobalvalue;
        bool isOptimizedDone;
        bool isFunctionReplaced;
        int  functionindex;
        int  globalvalueoffset;
            Patch()
                {
                     llvmGlobalVariableStartMachinePointer=NULL;
                     llvmOptimzedFunctionPointer=NULL;
                     llvmGlobalVariableEndPointer=NULL;
                     llvmFunctionPointer=NULL;
                     codePointer=NULL;
                     meth=NULL;
                     Size=0;
                     cusotmizedFor=NULL;
                     previousexetime=0.0;
                     numberofTime=0;
                     avgexetime=0.0;
                     previousglobalvalue=0;
                     isOptimizedDone=false;
                     isFunctionReplaced=false;
                     functionindex =0;
                     globalvalueoffset=0;
                }
} Patch;
```

- ***llvmGlobalVariableStartMachinePointer*** - Since each function has a linkage with corresponding global counter function, the emitted machine code memory location will be stored here.
- ***llvmOptmizedFunctionPointer*** - Current enhancement only support two level of optimization such as no optimization and full optimization. Initially all the code will be passed with no optimization and during the monitoring all the functions shall be passed with full optimization. Memory location of the newly emitted optimized machine code will store in this pointer.
- ***llvmGlobalVariableEndPointer*** - This is basically same as the previous llvmGlobalVariableStartMachinePointer which stores the end global counter function's memory location.
- ***llvmFunctionPointer*** - First time Java byte code function converted into LLVM IR representation corresponding llvmFunctionPointer will be returned to an application. This will be used to in the later case whenever we need new optimization.
- ***CodePointer*** - This pointer stores the location of the machine code address where first time function has emitted into machine code. This memory location will be utilized during the memcpy with a new optimized function.
- ***Meth*** - This is stores each Java method pointer which contains all the local variable list , dependencies etc.
- ***Size*** - The actual size of the machine code .

Reset of the fields are used to calculate the function call frequency as shown in the following pseudo-code

```
while (true) {
  loop through the patchMap
     {
     lock();
     globalVariable = get the globalvalue;
       if (globalVariable > 0
        {
         set_previous_global_value = globalVariable;
         execution_time  = current_time-previous_time;
         accumulating_the_execution_time += execution_time;
         previous_time = current_time;
        }
     unlock();
     }

 int l_timegap = current time-m_profilerstarttime;
  if (l_timegap > 20 second) {
    m_profilerstarttime = rdtsc();
   loop through the patchMap
        {
        lock();
   int samplingfrq = current_global_counter-
                     previous_global_counter;
        double frequency=0.0;
        if(samplingfrq !=0)
    frequency = accumulated_execution_time / samplingfrq;
        if (isOptimizedDone) {
  PatchmaterializeFunction(codePointer,
            Javameth,cusotmizedFor);
        isOptimizedDone= true;
       }
    previous_execution_time = current_time;
    accumulated_execution_time =0;
    EndglobalValue =get_the_end_global_variable;
    if (current global variable == EndglobalValue
      && isOptimizedDone && !isFunctionReplaced)) {
        isFunctionNotReplaced = true;
     memcpy(codePointer,llvmOptimzedFunctionPointer,Size);

          }
        unlock();
      }
   }
   sleep(1);
}
```

Every 20 second interval function frequency will be calculated by using previous information gathering. After the 20 second , optimized function will be put into the memory when the following conditions reach.

- *If that function has not been placed before*
- *If both start global counter and end global counter is same*

5.4 Gnuplot Thread

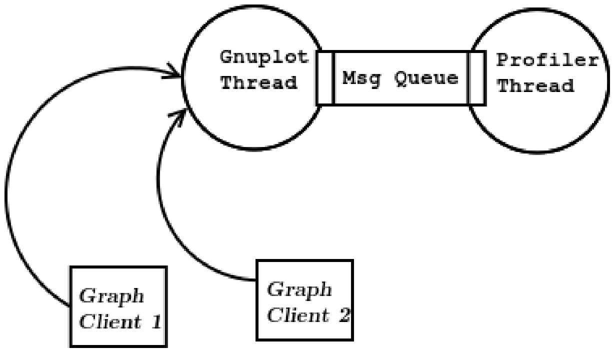


Figure 9: Thread communication in between gnuplotthread and profiler thread

It would be an additional advantage of having real time graph plotting when the function are executing in the background. We could be able to see what functions are currently being invoked and the relevant statistics such as average execution time, global counters and function call frequency. Actually, embedding this features inside the VMKit would break the actual purpose of VMKit and make an extra level of overhead to upper level VM. We have overcome this issue by implementing separate thread to disseminate graph message for all gnuplot graph clients whenever they connect to VMKit. All the produced profile messages are transmitting through the message queue channel and it will be sent over the network when client is connected to some dedicated port. Figure 9 shows how thread communication has performed inside the VMKit.

6 Evaluation

All of our experiments used identical hardware and software configurations: a single 2.4 GHz Core i3 with 4GB of memory and running a Linux 3.8.0-39-generic kernel . In order to show how well adaptive optimization perform over non adaptive mode execution, we have carried out the following evaluation.
Java source file has been composed with several functions such as heavy factorial calculations and matrix multiplications.

The following graph is showing the X86-64 machine code size in two different modes which are non-optimized and optimized. After we applied several LLVM passes , optimized code has yielded highest optimized code and size of the machine code is significantly very low. When the program startup non optimized machine code will be executed to reduce the initial boot up time and when the time goes fully optimized code will be patched during the runtime. One of the function(Calc1) has been chosen among other

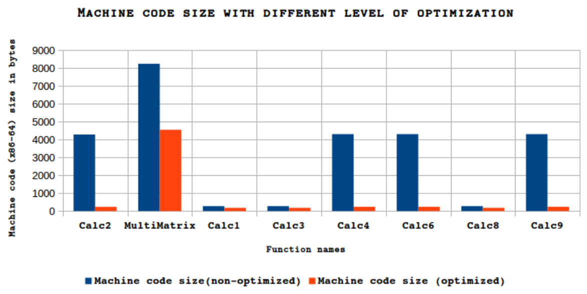


Figure 10: Machine code size with different level of optimization

functions to compare the non-adaptive and adaptive mode performance during the execution of the program. Every 20 sec interval the following calculations are being carried out.

- Function call frequency - How many time particular function has been invoked in one sec
- Global counter deviation - The difference of global counters during the 20 sec time interval.

Optimized Calc1 function has been replaced in the memory after 20 sec and we can clearly see the significant improvement on the rest of adaptive mode execution. Once the optimized code has placed into the memory the deviation of global counter and frequency have increased. We gained this performance moderation, because the optimized code is always running after 20 sec.

Another function MultiMatric which had the highest processing method among others have chosen to analysis our two type of approach. According to our previous machine

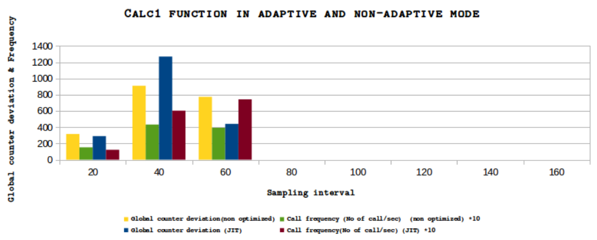


Figure 11: Modified components in existing VMKit

code size graph observation the size of MultiMatric has reduced to half of the size of non-optimized code, even though such a reduction is possible in the MultiMatric function, the following graph didn't show any significant performance improvement during the runtime. Why ? Calling frequency of this function is comparatively very low and we couldn't able to see the significant improvement over non-adaptive mode of execution.

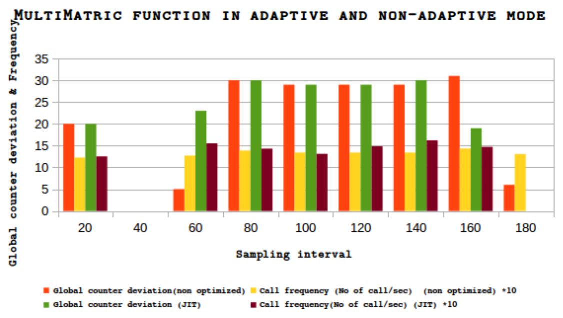


Figure 12: Modified components in existing VMKit

Finally we have moderated the given user input file with different loop count in order to extend the program running time so that adaptive mode of execution will take an advantage of that. The program has started with loop count 2000 and inside the loop different combinations of the functions have been called , It will give an opportunities to other function to compile with new optimized form then will be replaced in the memory. Blue line indicates the non-adaptive mode of execution and it almost linearly grows with number of loop count and orange line denotes the adaptive mode of execution and it also grows with loop count but with slowest rate. If we use different level of optimization in the adaptive mode , we could be able to see the wide deviation in the execution time.

7 Future Work

Even though we could be able to demonstrate an adaptive approach in VMKit with LLVM-JIT compiler, still there are

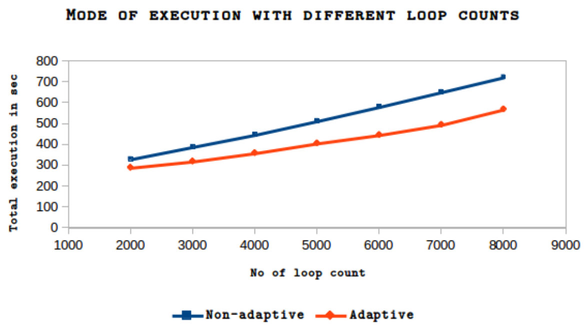


Figure 13: Modified components in existing VMKit

some functionalities lacking in our proposed method such as the following.

- Instead of modifying Java byte code to hook our own instructions , we can do the same modification on LLVM IR
- If the function is not invoking more frequently after we patched that function, we can degrade the optimization level and patch it again
- Handle garbage collection for all optimized functions, currently there are some issues(Segmentation fault) in handling optimized function.

8. Conclusion

VMKit provides a substrate for the development of VMs within a short period of time but still it has left interesting research areas in optimizing the execution of the VMs. Therefore our proposed solution for the LLVM JIT improvement would be an exciting research area to identify and exercise Virtualization concepts in practice. Because the amount of documentation on this VMKit, LLVM is so scarce, most of the considerations we presented come from code analysis. Overall, the code we produced does not have many lines, nor did it take much time to write, instead most of the time on this project was spent on reading through the source code of the VMKit architecture and LLVM IR generation and trying to find out how to best handle the data structures or which was the best location for the developed functions. We have contacted thorough llvm IRC channel for further clarification and gained more information about LLVM functionalities.

References

[1] Jikes RVM (Research Virtual Machine). `http://jikesrvm.org/MMTk`.

[2] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[3] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Virtual Execution Environment Conference (VEE 2010)*, Pittsburgh, USA, March 2010. ACM Press.