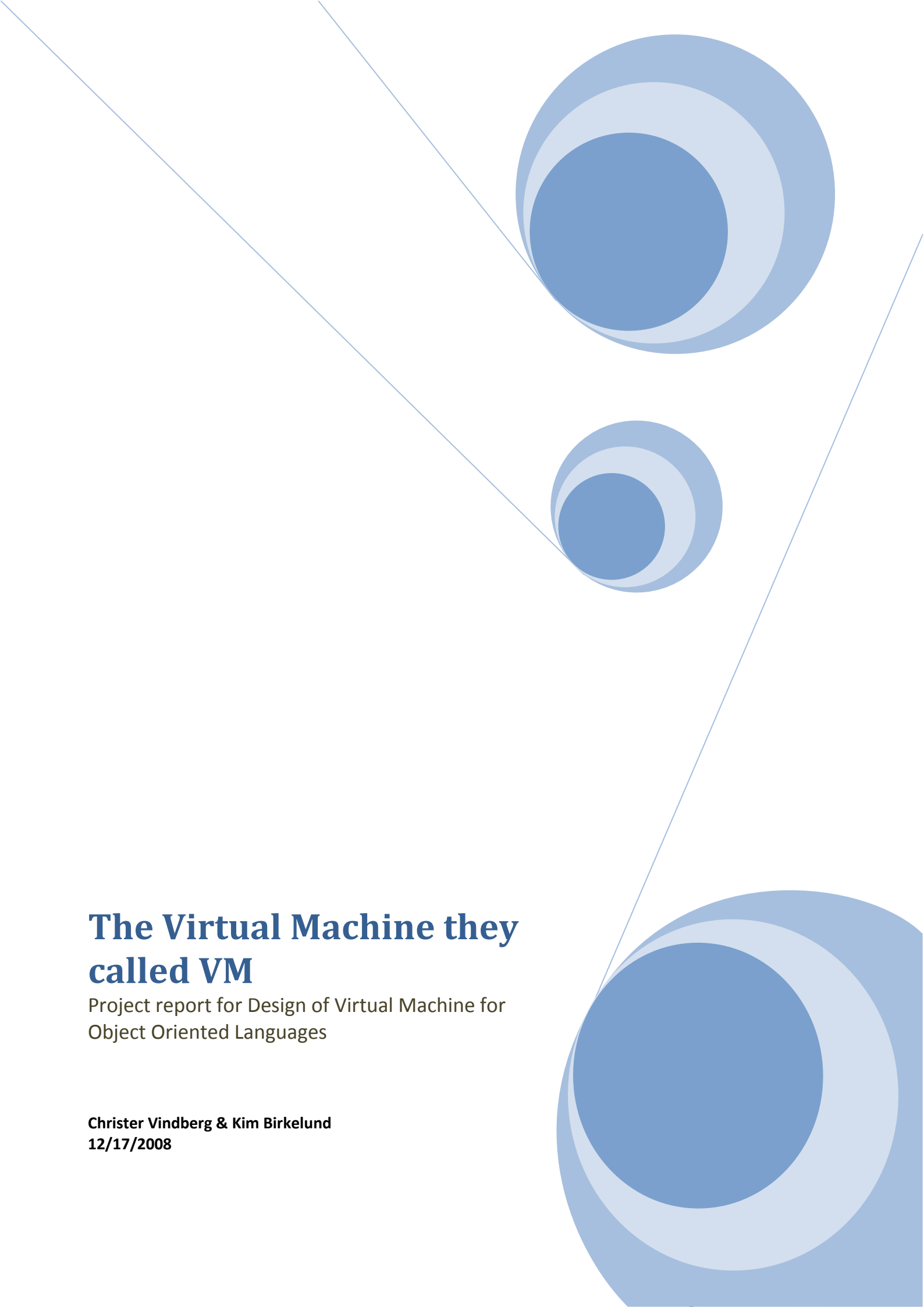


The Virtual Machine they called VM

Project report for Design of Virtual Machine for Object Oriented Languages

Christer Vindberg & Kim Birkelund
12/17/2008

Table of Contents

Status

VM language - VMIL

We decided to design our own language, not only does that allow us to fit it to our Virtual Machine (VM) or vice versa, it also means we had a very limited amount of opcodes. So we have implemented and tested all of them. If we were to continue working on the VM more would likely be added so that we can support more features.

Currently our language, which we will describe later, is very simple. We need to write 4 lines of code just to add two numbers. This, while not limiting the power of the language, seriously limits the productivity, it is very hard to code in and the number of lines needed for even simple programs is gigantic. You constantly have to make sure you do not mess up the stack and sometimes you need to push object, such as the console, unto the stack way before you need it. So we really need are more high-level version of our language to abstract away all these difficulties.

We have also, because of limited time, implemented a lot of functionality in C# and then used external functions to call them; we would have wished that we had time to implement more of it directly in our language.

The following are the opcodes comprising our language.

- *load-field <field-offset>*
  Loads the field at the given offset onto the stack.
- *store-field <field-offset>*
  Stores the value on top of the stack into the field at the specified offset.
- *load-local <local-variable-offset>*
  Loads the local variable at the given offset onto the stack.
- *store-local <local-variable offset>*
  Stores the value on top of the stack into the local variable at the given offset.
- *push-literal <string-or-integer-literal>*
  Pushes the specified literal of type string or integer onto the stack. Internally this is converted to specific opcodes depending on the type of literal.
- *pop*
  Discards the top stack element.
- *dup*
  Duplicates the top element of the stack.
- *new-instance*
  Expects to find the name of a class at the top of the stack which is then resolved into the actual class. A new instance of that class is then pushed onto the stack or a class not found exception is thrown.
- *send-message*
  Sends the message located on top of the stack (in the form of a string) to the object found argument count elements further down the stack. Messages are always of the form *<name>:<argument-count>* and so the number of arguments can easily be found prior to message

handler resolution – actually the number of arguments is part of the handler name thus allowing overloading.

- *return-void*
  Returns with no value.
- *return*
  Returns the value at the top of the stack.
- *<label>:*
  Specifies a new label.
- *jump <label>*
  Uncontional jump to label.
- *jump-if-true <label>*
  Sends the message *is-true:0* to the object at the top of the stack and jumps depending on the values returned (greater than 0 is true, everything else is false). Null at top of stack implies false.
- *jump-if-false <label>*
  As above but the message sent is *is-false:0* and null at the top of stack implies true.
- *throw*
  Throws the exception at the top of the stack.
- *.try { <instructions> } catch { <instructions> }*
  Normal try-catch-clause. Unlike the Java or .NET VM we recorded entry into try-catch block with special stack values.

Following are two examples of programs written in our language. First we simply add two numbers.

```
push-literal 1
push-literal 1
push-literal “add:1”   //(1).add(1)
send-message
```

A slightly more complex example is getting an element from an array and printing it. We have stored the console instance in a field called *con* and the array in a field called *arr*. The object representing the console is stored in a variable as we have no static members.

```
load-field con
load-field arr
push-literal 1
push-literal “get:1”
send-message
push-literal “write-line:1”
send-message
```

In C# this would be something like: `System.Console.WriteLine(arr[1])`.

The syntax of the language is described in the following in an informal grammar. A precise grammar for the language can be found in the source code.

```
<class-declaration> =
  .class <visibility> <name> extends <full-class-names> {
              .fields { <names> }

              .handler <visibility> <name>( <names> ) {
                          .locals { <names> }
                          <instructions>
              }

              .handler <visibility> <name>
                          .external <external-name>( <names> )

              .default {
                          .locals { <names> }
                          <instructions>
              }
              <class-declarations>
  }
<visibility> = public | protected | private
<name> = normal C style identifier
<full-class-name> = <name>(.<name>)* // Full path to class
<external-name> = <name>(.<name>)* // Path to method in SystemCalls
```

Since we have a very limited opcode set many common functions such as integer addition and subtraction are external library functions. This limits the number of functions we can implement directly in our language and also impacts performance. Since so many of the basic functions are external, we often need to switch from the interpreter to a .NET method. Currently that probably does not hurt performance that much but if we wanted to do JIT compilation and inline-caching it likely would since these methods cannot easily be inlined. For many of the most basic operations it would be beneficial if we could inline them in the interpreter – like recognizing the message *add:1* if two integers are found on the stack.

The currently implemented API is actually quite large considering the nature of the project and a listing can be found in Appendix A.

Status compared to initial plan

Initially we laid out six milestones, with the intention of completing one per week. This turned out to be very unrealistic, since none of us had implemented a VM before. Even after one week it was clear that the time table was slipping. We did however manage to complete the first three milestones. The last three which we described as “nice-to-have and not must-have” were only partly completed.

Milestones 1, 2 & 3

M1 said that we should be able to execute most of our language and M2 said to support the full language and a directly loadable binary format. We initially had a binary format but dropped it later on, however we do support the full specification of our language. M3 said simple stop-and-copy GC. We did manage to implement a working GC but we decided to do mark-and-sweep instead of a copying-collector.

Milestone 4

In M4 we had planned to do a generational collector and while we did implement a GC it is not generational. It would be interesting to see how our VM perform under different GCs but we ran out of time and never managed to make that comparison.

Milestone 5

As M5 we had planed “JIT-compiler, may not include inline-caching”. We never got around to looking into JIT-compilation but it would without doubt have helped with performance, the same goes for inline-caching.

Milestone 6

The final milestone we had planed was “.NET interoperability, threading, inline-caching, trace-based compilation, compiling from gbeta”. We did manage to include full support for threading, but neither inline-caching, trace-based compilation, compiling from gbeta or .NET interoperability where possible within the time frame.

Not in initial plan

Besides the things we planned to implement according to the plan, we also managed to implement a neat feature that allows us to change interpreter at run time, meaning we can swap in a debug interpreter mid program. We did unfortunately not manage to finish the debugging framework.

Running the virtual machine

We have used Google Code as our repository and the source code along with a precompiled set of binaries can be checked out from http://dovmfool-vm.googlecode.com/svn/trunk/.

This repository contains several folders:

- **VM**
  Contains the source code for the entire project including support libraries. The project is made in Visual Studio 2008.
- **VMILTests**
  The test programs we have used to make sure the VM worked correctly. Obviously not comprehensive enough. A simple PowerShell script adding a few aliases is also included.
- **vmshl**
  Binaries for a release build of the project. The binaries are automatically copied to this location on compilation (both debug and release). The debug build contains several assertions to ensure integrity of the heap. The main executable is VMShell.exe. See next sub-section for usage.

The project is built on and requires .NET 3.5 which will likely already be installed on an updated Windows machine. The parser for our homemade language was implemented using the Garden Point parser and lexer generator found at http://plas.fit.qut.edu.au/projects/. The version we built against is included in **VM\GPPG_GPLex_Bin**, and it appears that our grammars are incompatible with the newer version available at the website.

Executing VMShell.exe

Executing the program with no parameters will print the usage but for reference here is the syntax:

VMShell.exe [-InputFile] <filePath> [-Pauser] [-Swapper]

The InputFile parameter can be given by position where as the Pauser and Swapper parameters must be given by name. Specifying either Pauser or Swapper activates one of two different test modes. The Pauser mode will pause and resume all interpreters periodically; this mode was created to test functionality needed for garbage collection. The Swapper mode is the one shown at the presentation and will simulate attaching a debugger, stepping 20 times on each thread and detaching the debugger again; this mode was to test that the preparation build for debugging functioned[1].

Architecture

The solution is separated into a seven projects:

- **Sekhmet**
  General purpose support library not written for this project. Used both at runtime and in the tool-chain used during compilation.
- **GPPG**
  Support library for the generated parser – written by the developers of the Garden Point Parser Generator. Used at runtime.
- **FamPoly.Utils**
  Tool created as part of a different project[2]. The purpose of the tool is to make a few changes to the code generated by the parser and lexer generator tools to make them fit nicer with the rest of the codebase.
- **VMILLib**
  A standalone library dealing with the source language. Originally it read and wrote both a textual and binary representation of said language, but as the project progressed the binary format was abandoned.
- **VM**
  The main virtual machine binary. This is built as a library that can be included in other applications.
- **VM.Debugging**
  Library implementing most of the interface for a debugger. The debug interpreter and execution stack is part of the main VM library but the Windows Communication Foundation interfaces and classes are implemented here. This library is not completed but part of it is used in the Swapper test mode.
- **VMShell**
  The current console based driver for our VM. Simple console based program that simply invokes the virtual machine based on the parameters given.

The static class *VirtualMachine* (**VirtualMachine.cs**) implements the interface programs wishing to use the VM. This interface is simply the ability to execute a given file in a synchronous and asynchronous manner (no callback is currently specified for the asynchronous case) as well as the ability to attach and detach a debugger. When requesting the execution of a file the VM is automatically initialized, resulting in the

[1] Debugging functionality has not been completed unfortunately.
[2] Part of Kim Birkelund's PhD project.

creation of a memory manager and the loading our base library, before execution is begun. Our base library is embedded in the binary but is written in the VM's source language with references to external methods[3].

Execution itself is performed by creating a new instance of *InterpreterThread* (**InterpreterThread.cs**) specifying an object instance and a message handler within defined by the class of that object. Each *InterpreterThread* instance represents a distinct thread and so creation of a new thread is done in the same manner though initiated from within the VM.

The *InterpreterThread* then creates a new interpreter using the currently specified *IInterpreterFactory* (**IInterpreter.cs**). This factory specifies methods for creating new *IInterpreter* (**IInterpreter.cs**) and *IExecutionStack* (**IExecutionStck.cs**) instances. The interpreter itself has a single entry point: *Run*. To ensure that the interpreter can be swapped at more or less any point we have an invariant specifying that only the *InterpreterThread* instance of a given interpreter is allowed to call this entry point. So to ensure that external method calls can call back into the VM a stack of messages stored in the thread is employed. All code has access to the *InterpreterThread* instance it is executing in and can add a new message to its stack – implemented as .NET delegates – allowing it to call either other external method for the interpreter.

A property in the interpreter ensures that it can be made to return when the currently executing instruction is executing. Note that due to the message stack any external call that the interpreter may initiate is actually relegated to be performed by the *InterpreterThread*. An interpreter running our own language can thus be paused almost instantly and any running .NET code will pause when returning from the most recently made call – if a thread is blocked it is simply specified that it should pause. Obviously this means that any path that can lead to blocking should be guarded on reactivation to avoid memory accesses when that happens.

To make access to heap objects easy from within the VM each type of heap object recognized by the VM has a corresponding .NET *struct*. Because this is a managed platform we cannot manipulate memory the way one would in C++ so these *struct*s have a single field representing their address in the heap. Access to the contents of these types is done through static methods. In reality access is gained through a generic handle parameterized with the intended type. To access the name of a class one would call the method *Name* with an instance of *Handle<Class>*. Through the magic of extension method in C# 3.0 this call can be made to look like a call to a normal instance method and the compiler simply rewrites it making the code look a lot nicer. The handle class allows us to have GC safe pointers by registering them centrally and updating them when moving objects[4].

As mentioned not all API function can be written in the VM language so external calls are necessary. Currently these are implemented in the classes nested beneath *SystemCalls* (**SystemCalls.cs** and **SystemCalls.*.cs**). When a reference is made to an external method named *System.Reflection.Class.Name* the VM expects to find a class .NET *SystemCalls.System.Reflection.Class* and in that class a method called *Name* with a signature matching the delegate *SystemCall* defined in **SystemCalls.cs**. Each class containing external methods are decorated with the *SystemCallClassAttribute* (**SystemCalls.cs**) attribute and likewise each method implementing an external call is decorated with the *SystemCallMethodAttribute*

[3] The syntax for these external references is part of the language and can be used by anyone.
[4] Currently we don't actually move any objects during GC, but the plumbing is in place to support it.

(**SystemCalls.cs**). This allows the resolution mechanism implemented in *SystemCalls* to find, and cache, external methods when they are first accessed.

Memory management is handled by the *MemoryManager* (**MemoryManager.cs**) class which acts as the manager of memory managers. This class can hold a number of child memory managers which each operates on a subset of a shared array representing the heap. Currently we only have two different memory managers: a mark-sweep collecting one and a non-collecting one. The idea was to let *MemoryManager* manage the various generations, but we never got that far.

Memory representation

As mentioned, the heap in our VM is represented as an array of unsigned integers[5]. Each object allocated on the heap has a two word header containing the size, GC bits and a class pointer. To overcome the bootstrapping problem of needing a pointer to the *Class* class before it has been loaded a number of "known" classes are bound to negative values during initialization and subsequently replaced when negative class pointers are encountered. The size recorded in the header is only used during GC and is not guaranteed to be the actual size of the object; see the section on our mark-sweep implementation. The pointer returned from the memory manager when allocating points to the first word after the header to make it easier for the actual object implementations to keep their offsets straight. Because the header is not part of the objects own representation it is also not part of the size given to the memory manager when allocating, and as such it would be illogical for the objects to have to take it into account when accessing their contents.

We have six different object layouts representing application defined objects, strings, arrays, classes and two forms of message handlers.

The two **message handler** representations share a two word header. The first word encodes a string pool index, visibility, whether it is an external handler and whether it is marked as an entry point. The second word contains a pointer to the class defining it. The representation for **external message handlers** takes up further two words: one for the external name used to resolve the actual .NET method and one for the number of arguments the method expects. **Internal message handlers**' takes up a lot more: one word for local variable, argument and instruction counts (eight, eight and sixteen bits respectively) and an additional word per instruction.

Classes contain a lot more information and thus have a more elaborate representation. The first word encodes visibility and name (as a string pool index), the second is a pointer to the parent class, the third points to a string containing the name of the file defining the class (to be used for debugging purposes). The fourth contain field and message handler counts and the fifth contain inner class and super class counts. The sixth word contains a pointer to the class' linearization if computed. From the seventh word forward the contents is guided by the counts in word four and five. First are the names of super classes, followed by message handlers and lastly inner classes. Because fields are only accessible from with a class there is no need to store their names as all loads are converted to offsets during class loading. Message handlers and inner classes take up two words each as we copy their header to make resolution faster.

[5] Technically we have a *Word struct* wrapping

The representation of **strings** is quite straight forward: one word for length in characters, one for hash code and one for every two characters[6]. Going forward we would have liked to make more efficient representations for substrings and concatenations but we simply did not have the time.

The **array** representation uses one word for the element count and two words per element to store a class pointer and object pointer. We store integers directly and so a way to distinguish between references and integers was needed. Our first approach allocated a number of words in the beginning of the array for a map indicating which entries were references and which were integers. This representation was doubtless more efficient but was too susceptible to race conditions: if two threads were to update two different elements that used the same word as map the risk of losing one of the updates is quite big. While we did not test it the overhead locking the array on each update would incur was deemed too great. Obviously a race condition can still occur if two threads update the same element but that is to be expected. Although the problems that can occur in our VM compared to e.g. .NET are a bit more serious. If one thread updates an element with an integer and another thread updates the same element to some object, it is possible that the integer will be treated as an object subsequently.

Finally we have the **application object representation**. The class of these objects is already recorded in the common object header so no need to store that once more[7]. Because we support multiple-inheritance[8] we cannot calculate the final field offsets locally for each class but need to do it for the specific class we are instantiating. But even though we can calculate the final field offsets when we know the exact class we cannot actually update the message handlers so instead we store an additional offset which will be added to the locally calculated offset on each field access. These offsets are stored in the beginning of the object[9]. The first word is used to store an offset to the first field and the second forward stores the aforementioned offsets followed by two words per field (one for class and one for object reference).

Overall the current design represents an attempt at making a tradeoff between generality and efficiency. Because we have several different object layouts we cannot uniformly scan each reference in an object, but if we had a completely uniform object layout we would have terrible locality in the layout of classes and message handlers. The object header could have been limited to one word leaving the class pointer to be part of the representation of application objects and identifying the rest using some hardcoded bit pattern in the remaining header word. This would limit the maximum size of objects but not by so much as to make it a problem – currently the maximum size is 2^{30} would be lowered to something like 2^{26} which is still more than enough to be acceptable. This layout did however allow us to more directly reuse the objects representing classes and message handlers as mirrors in our reflection API. Generally any object on heap can be treated like an *Object* (the base type in our language) with no special cases.

---

[6] Conversion from .NET strings to VM strings are done by the built Unicode encoding found in *System.Text.Encoding.Unicode*.

[7] An interesting side note is the fact that because the class stored in the object header is determined by the type parameter given to the allocation method and thus the class of application objects cannot be recorded this way. Instead the class is updated by the helper method used to create application object instances.

[8] And because super classes are not resolved until the first instantiation.

[9] This also represents a bit of wasteful object layout design: the list of offsets might as well be stored in each class instead of in each object.

Garbage collection

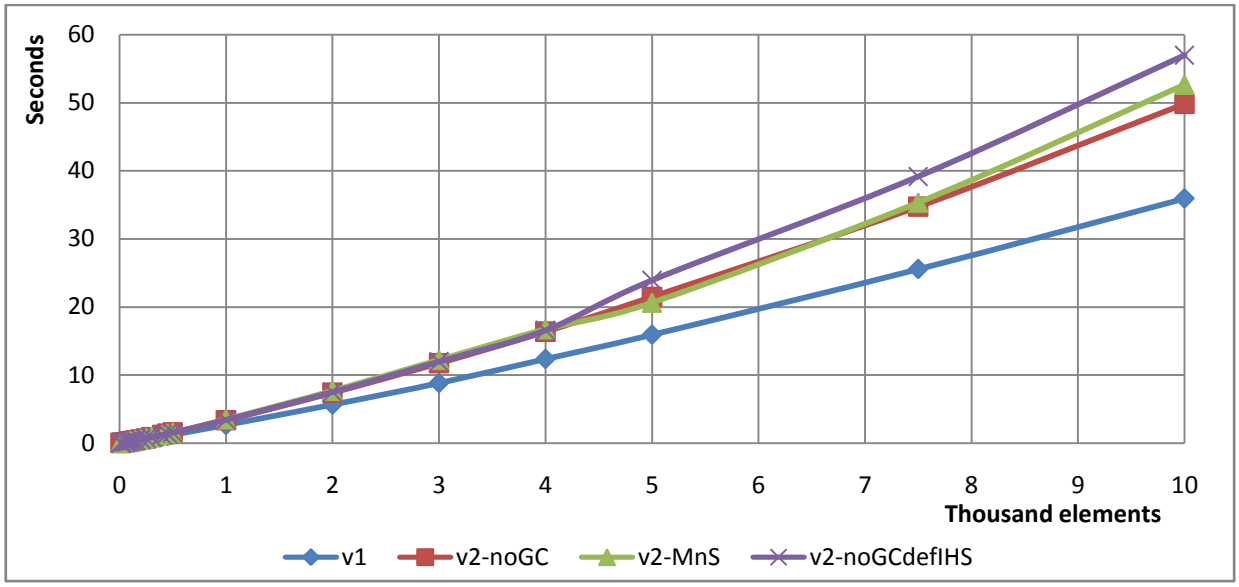
After running for a long time with a simple non-collecting memory manager we finally managed to get a mark-sweep implementation working. The layout of objects basically stayed the same as it was already designed to support GC. The thing that did change is that recorded object size can be larger than requested when allocating. This is due to the way we track holes in the heap. Each hole records its size in the first word and a pointer to the next hole in its last word. Because the common object header takes two words any useful objects takes up at least three words and so a hole of length less than three is useless – at least until one of the objects around it is collected. Instead of waiting for the death of a surrounding object any time a whole of size less than three is created we simply treat it as though the allocation filled the entire hole. The reference from the previous hole is thus updated to jump over the eliminated hole. However because the recorded size of an object is used to scan the heap during sweeping we need to pad the object just allocated with the size of the hole making it one or two words longer. Not realizing this from the beginning caused a lot of problems.

The root set is quite easy for us to obtain. Firstly we take all of our handles which are recorded centrally along with any references stored on the stacks of any running interpreters. Since we use two words per stack element (class and pointer) sorting integers from objects is quite easy. To make updating of handles and stack elements easy when moving objects a dictionary is created mapping each root to a delegate making the appropriate update. Once this dictionary has been created it is given to the marking phase. Running through each root the marker extracts the object type and calls a method that return all the references in that object ensuring that all reachable objects are marked. The allocation, marking and sweeping implementations themselves algorithm itself are straight forward.

While we did not have time to implement compaction (making our ability to actually tracking object moves pointless) we did manage to implement heap expansion. Currently, if an allocation fails, a garbage collection is initiated and the allocation is retried. If it still fails the heap is expanded and we retry again. In the end if the maximum specified heap size is reached and the allocation still cannot be made an out of memory exception is thrown.

Benchmarking

We decided to implement a simple merge sort to test our VM. With merge sort we could test how the VM would perform with a very CPU and Memory heavy program. We ran the tests on several different versions of thee VM, the one we had at the presentation which did not have a GC, our current implementation with the GC turned off but with a very big start heap size, with the GC turned off but a small start heap size so it would have to expand the heap to accommodate the program and last with GC turned on.



In the graph above v1 is the version demonstrated at the presentation. V2-noGC is the current version with garbage collection disabled and an initial heap size large enough to hold the entire benchmark. V2-MnS is the current version with mark-and-sweep garbage collection and dynamically increasing heap size starting at 10.000 words and the last, v2-noGCdefIHS is like v2-MnS but with GC disabled.

The first thing to note is that the current implementation is slower, this was not that surprising since we have added features to the VM and changed the way it handle Handles and not done a single thing to optimize it. It is good to see that the GC version performs almost as well as the Non-GC version. Finally the Non-GC version with the low start heap size has to do a lot of expanded to be able to contain all the generated memory allocations to its predictably slow.

We also tested our VM against other systems, C# and JavaScript, to see how well it would perform compared to those. The results we got was however incomparable. While our VM took almost ten minutes to sort 100.000 elements C# and JavaScript sorted 1.000.000 elements in half a second and two seconds respectively.

We knew it would be slow but maybe not that slow. Part of the reason is that many of the basic instructions such as Add, Subtract and so on are external library functions which are expectably slower than if we had implemented them direct as opcodes. It would have been interesting to see how fast we could have made it run with more basic opcodes and optimizations such as inline-caching, JIT. But as noted in our plan we never attempted to focus on performance, we were more interested in building a general VM since we never done that before. That said it would have been nice if it had been a bit faster.

Conclusion

While the VM we have implemented does not really implement the techniques that have been at the core of this course it believe it has been very productive. Our starting point was not knowing anything about the construction of a VM from the ground up, and thus we were not able to apply the techniques taught. Now we know how to build a VM and can see how the techniques we have learned can be applied. Obviously we

still have a long way to go before we can create high performance VM able to beat V8, but we are closer to that ability than when we started the project.

As for the VM itself we would have liked to expand on the garbage collector. Because we stopped prior to implementing compaction we have not had to actually update memory references in live objects and because we only have one generation we also have not had to deal with references from older to newer generations. These are two issues that are non-trivial to get right and some experience in their implementation would be nice.

Generally though we are very pleased with the amount of stuff we got working. It is always nice to see that design decisions made early on are not complete invalidated later. From a language design point of view we would probably do a lot of things differently (unifying classes and message handlers, eliminating fields and streamlining visibility) but the fact that the language actually works with all intended features gives a nice sense of accomplishment. Seeing that first program run through without barfing is something one could get hooked on.

Appendix A

API

Class	Operations
Array	*new-array(size)* *get(index)* *set(index, object)* *copy(srcindex, srcarray, destinationindex, destinationarray)* *copy-descending(srcindex, srcarray, destinationindex, destinationarray)* *index-of(element)*
List	*initialize(initialSize)* *add(object)* *length()* *index-of(object)* *get(id)* *insert-at(index, object)* *remove(object)* *remove-at(id)*
Integer	*to-string()* *subtract(other)* *add(other)* *multiply(other)* *divide(other)* *modulo(other)* *compare-to(other)* *negate()* *is-true()* *get-hashcode()*
String	*get-hashcode()* *equals(other)* *compare-to(other)* *substring(start, count)*

	split(splitter) *index-of(substring)* *last-index-of(substring)* *concat(otherString)* *length()*
Object	*get-type()* *to-string()* *equals(other)* *is-true()* *is-false()*
Console	*write-line(str)* *write-line()* *write(str)* *read-line()*
Exception	*initialize(message)* *to-string()* *message()*
Visibility	*initialize(vis)* *is-public()* *is-protected()* *is-private()* *is-none()* *to-string()* *equals(other)*
Threading	*start(obj)* *join()* *run()* *sleep(milis)*
Class	*name()* *visibility()* *parent-class()* *super-class-names()* *super-classes()* *default-message-handler()* *message-handlers()* *inner-classes()* *equals(other)* *full-name()* *to-string()* *trace(console)* *trace(console, indent)* *pad(console, indent)*
MessageHandler	*name()* *argument-count()* *visibility()* *is-external()* *is-default()* *equals(other)* *to-string()*