Runtime System Specification

Pat Caudill
Instantiations, Inc.

March 5, 1998

This is the original design document describing the architecture of the runtime environment targeted by Instantiations' JOVE optimizing compiler for Java. The architecture includes object representations, garbage collector design, threading and exception handler issues, stack frame design, calling conventions, and register usage conventions.

This runtime was designed by Allen Wirfs-Brock and Pat Caudill with input and review from the rest of the Instantiations' technical staff.

This document represents the design as it was envisioned fairly early in the development of JOVE. Some aspects of the design evolved over the course of development of the two major releases of JOVE. I have added a few notes to the document commenting on some of the changes. Other than those notes, this document is as originally written by Pat.

The garbage collector design has its roots in the multigenerational collector developed for Tektronix Smalltalk (see OOPSLA 86 paper). The design concepts had been subsequently refined in several Digitalk Smalltalk VMs. This was our first attempt to apply the techniques to a (relatively) static language and generally we were quite pleased with the performance of the collector.

Allen Wirfs-Brock
February 2011

Object Format

The format of objects is designed for minimal storage consistent with fast execution. We will separate large objects from smaller ones and objects which may be shared between threads from those that are local to a single thread. All fields, except the header are optional in any given object. The object layout is given in fig. 1

Object Format0

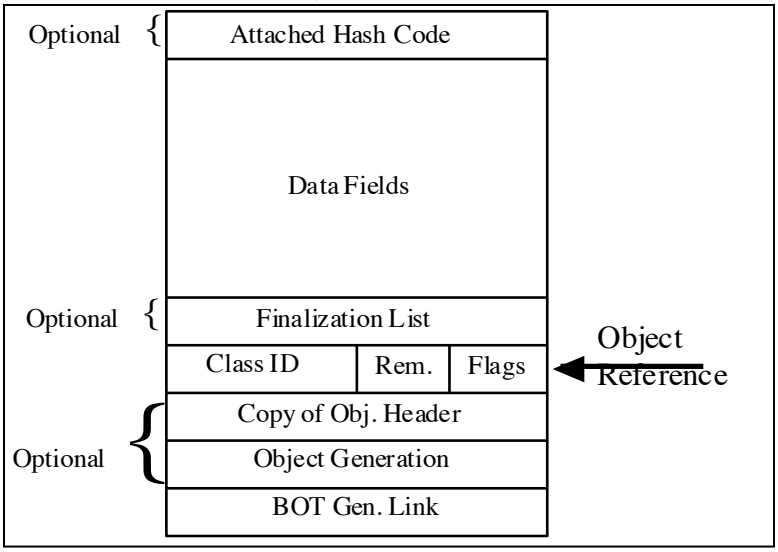

Fig 1.

The Class ID, remembered flags and Flags together are referred to as the "Object Header." All references to an object are through a pointer to the Object header. This is a pointer to the Flags field on little endian machines and to the Class ID on big endian machines. The Object Header is always addressed at a memory location divisible by four.

> AWB 2011: JOVE was only implemented for the IA-32 (x86) architecture so endian issues were never actually addressed.

All fields except the Object Header are optional. The Data Fields are of varying numbers of words with the exact number dependent on the Class ID, or for arrays on the length. An array object will contain uniform data, either binary or object references, in either case all data will be stored after the length field such that fields with higher indexes are stored in higher memory locations.

The Class ID field is a 16 bit field which contains an index number identifying the the class of the object. This field may be used to index into tables to determine various information about the class. This data may be kept as individual data or as separate tables for each piece of data. Examples of the data indexed by the class ID are the message look up tables, routines to give the object size in bytes, ... All arrays of object references will have the same class ID. These arrays will have, as the first word following the length, an object reference for the class object for this array. this will allow us to dynamically create new kinds of arrays without growing the tables indexed by the class ID.

AWB 2011: Using a class ID meant we didn't need to have a full sized pointer to a class descriptor or vtable in each object. This is how we got a 4 byte header. It's still as efficient as a vtable because of our use of inverted selector table row displacement dispatch. We didn't have to worry about garbage collection of the tables referenced by the class ID because JOVE was a whole program compiler without dynamic code loading. Hence all class metadata could be statically allocated by the code generator.

The Flags field contains bit encoded information about the object. The actual meaning of the flag bits is defined in fig. 2. The flags have been defined such that an object header is distinguishable from an object reference by examining the low order two bits. See the section on Memory Spaces for more detail about the remembered flags which are used by the garbage collection routines. They are tested and set during store operations.

Object Header Flags

Bit(s)				Meaning
0-1	=	00	=	This object has been forwarded.
	=	01	=	This object has no hash code.
	=	10	=	This object has had it's hash value referenced. the next time it is copied the hash value will be attached.
	=	11	=	The hash value is attached to this object.
2	=	0	=	This Object is small.
	=	1	=	This Object is large
3	=	0	=	This object does not own a monitor.
	=	1	=	This object does own a monitor.
4	=	0	=	This object is moved by the Garbage collector.
	=	1	=	This object is not moved by the Garbage collector.
5	=	0	=	This object is not an array.
	=	1	=	This object is an array.
6	=	x	=	reserved
7	=	0	=	reserved must be zero.
8-15	=		=	Remembered flags for this object.

Fig. 2.

The Finalization List field only appears in those objects which have a nontrivial finalization method. A trivial finalization method is one which does not perform any processing, or only sends to super where super is a trivial finalization method. The class Object has a trivial finalization method. The presence of this field is statically determined at compile time and is included in the run time size of the object. This field contains a pointer to another object but is not processed during the initial garbage collection processing. Until the collector notices that this object is no longer referenced this field is the link for a linked list of objects that will need finalization. After the collector determines that this object is ready for finalization this is the link

for a list of objects to which the finalize message must be sent. This field will be present/absent from all instances of a given class.

The Data Fields are of two types, object references and binary data. Object reference fields contain the data for those instance variables in the object which are typed as other objects. The actual data is a pointer to the object header of the value object. Object reference fields must be aligned to a four byte boundary. The Binary Data Fields contain the data for those instance variables in the object which are typed as binary data (e.g. int, float, ...). Binary data must appear on a "natural" boundary and fields may be padded by the code generator to ensure alignment. The instance variables defined in a given class may appear in any order as assigned by the code generator. The instance variables for the superclasses appear closer to the Object Header than the fields for their subclasses and are in the same order as for the defining class. For garbage collection purposes, the code generator will be required to generate for each class a routine which will extract all and only the fields which are object pointers and pass them to a routine within the collector.

> AWB 2011: The GC did a table lookup using the class ID to access the traversal routine for each object.

The Attached Hash field contains the hash value to be returned by the hashCode method in Object. This field is not created until the object has had the hashCode method sent. The hash value is determined at that time and is later attached to the object when the garbage collector moves it. The hash value will be attached to the object, the next time it is copied, if the low order two bits of the flags field contain the value 10_2. The presence of this field is indicated by a value of 11_2 for the low order two bits of the flags field.

The BOT Generation Link, Object Generation and Copy of Object Header exist only for objects which are larger than an implementation defined size. These objects are stored outside of the object space and have their object header large object flag set. The BOT generation link is the link field for a list of all the large objects which exist in a given generation. The copy of the object header is a scratch word used by the garbage collector during the collection process and is unused at other times. The generation is the generation that this object is logically part of. The generation number is a small part of the word and the rest may be reserved for other data. (By proper fiddling can I merge the object header copy and generation word?)

Memory Spaces

To improve locality of reference and to make the pauses due to garbage collection shorter, the memory for objects is broken into distinct areas called generations and segments. A generation is a collection of objects with about the same total lifetime and thread locality. An object is first allocated in the memory for generation zero associated with it's local thread. As the object survives garbage collections it is progressively promoted to higher numbered generations until either it dies, it ends up in the highest thread local generation, or it is shared.

There is a separate set of generations for objects that may be referenced from more than one thread. This will allow garbage collection of the objects local to a thread without disturbing the running of other threads. An object is made share able if it is stored into a static variable or an object that is itself shared. When this occurs the object and it's referents are transitively transferred into the youngest shared generation and the references to them are updated to the new

address. The memory for the old objects is then recovered on the next thread local garbage collection cycle.

> AWB 2011: The actual implementation evolved to defer copying the object until a GC for its containing segment occurred. At that point non-shared objects would be copied out of the segment by the normal GC process. Shared objects were left in place and the segment was reassigned to the shared generation.

The generations are not of fixed size. Each is composed of a set of segments which are dynamically assigned to the generation. Each generations segments form several linked lists with each growing by adding more segments as needed. During a garbage collection cycle, for a generation, the live objects in the generation are copied to a new set of segments and the old segments are recycled. Each segment is a power of two in size and is aligned to an even address of that size in memory. This allows the segment to be accessed by clearing the low order bits of an object's address.

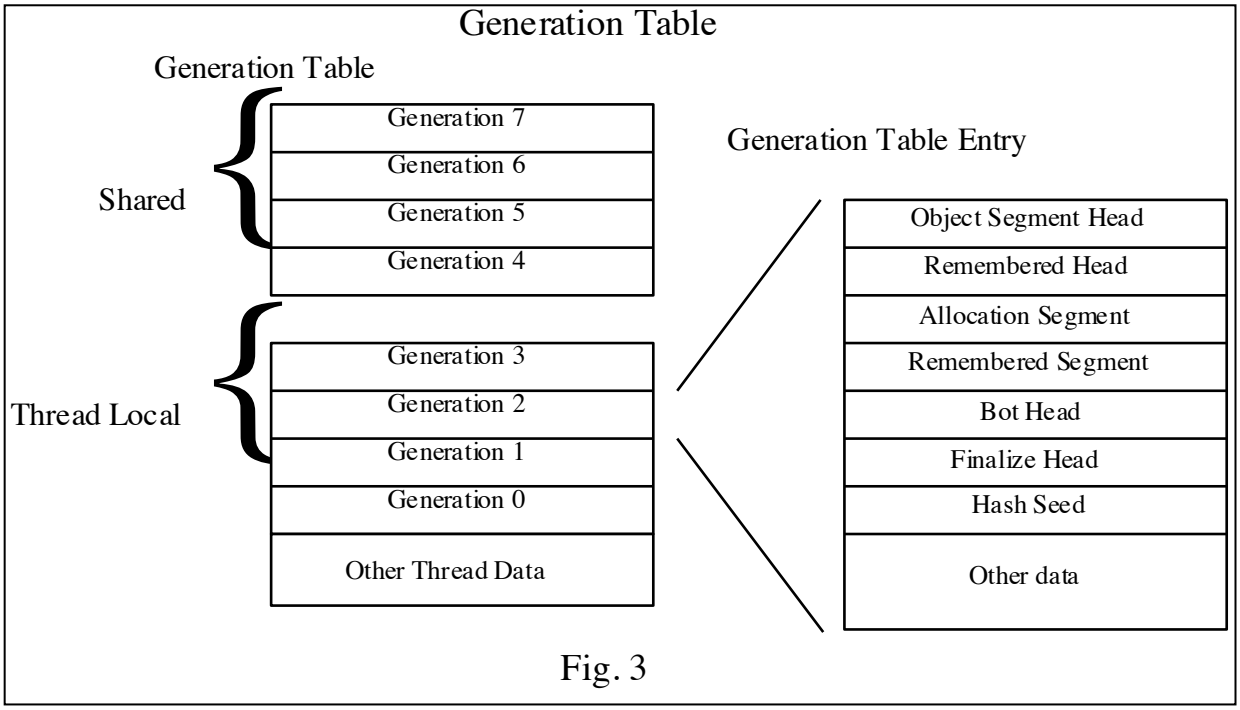

Fig. 3

The segments are be used for several different purposes. This is not strictly necessary but is done to simplify segment allocation. Object segments hold small objects which will be moved during each garbage collection. The segment has a pointer (allocation Pointer) which points to the first available slot for a new object to be stored. When the garbage collector copies objects into a segment it sets a pointer (Old Object Pointer) to the end of last object that it copies into this segment. This is used in making decisions about promoting the object to an older generation. An object segment also contains the generation of the objects stored within it. This is used during store operations and during garbage collections to maintain information about cross generational references.

The generation table entry contains a pointer to the start of the list for it's generation. It also contains a pointer to the first segment which is to be checked for available memory during object

allocation. The list is followed from the allocation pointer until a segment is found with enough memory or the end of the list is reached. If the end of the list is reached then a new segment is assigned to this generation and it is used for the object. When the amount of free space in a segment falls below a minimum the allocation pointer in the generation table is advanced to the next segment. When the number of segments reaches a limit, set dynamically, the generation is scheduled for a collection cycle.

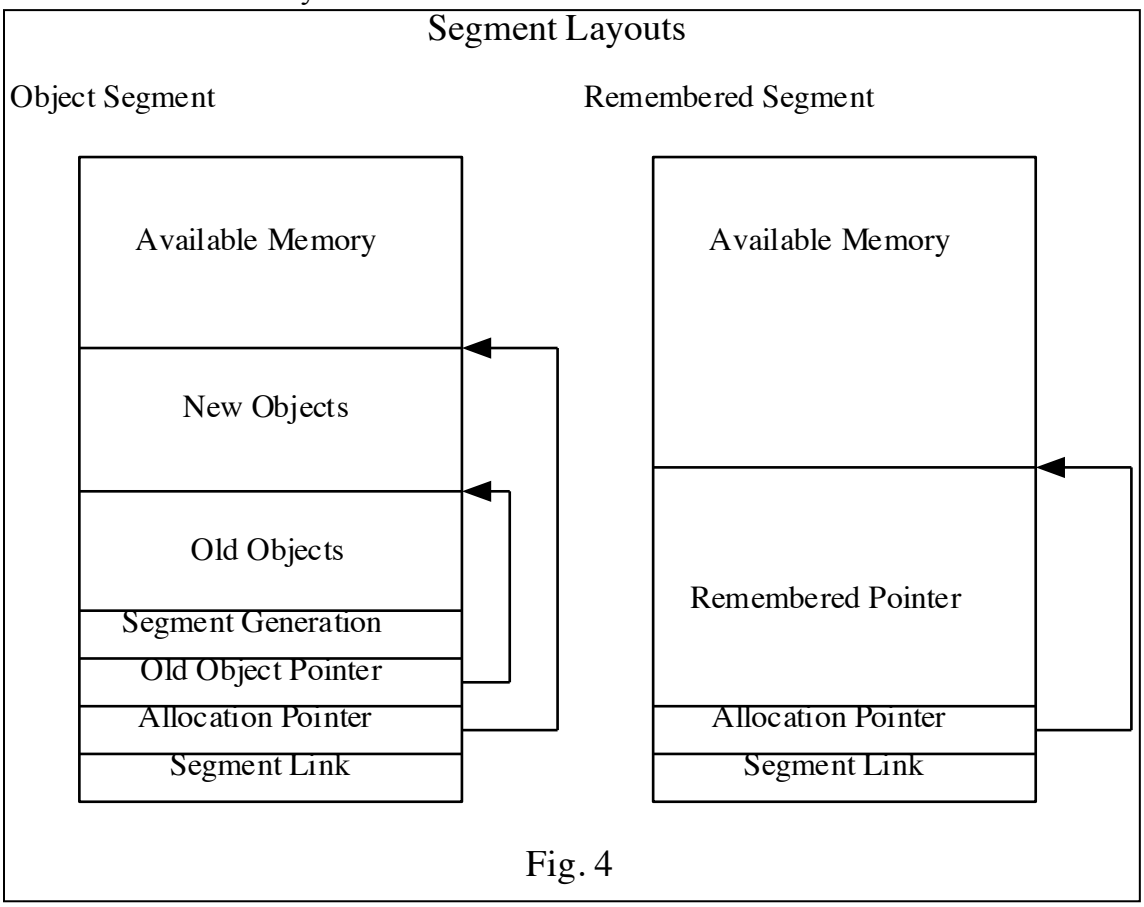

Fig. 4

It should be obvious that there is a fixed maximum size for objects in a segment. The largest object that will be allocated in a segment is actually much smaller than the maximum since we do not wish to copy very large objects during each collection cycle. The large objects are allocated in a separate area of memory and linked together using Big Object Table blocks. The BOT elements are shown in fig 1. with the object format. There is a separate BOT table for each generation and the head of the list is held in BOT head in the generation table entry.

AWB 2011: To clarify, there isn't an actual table of big objects, but instead each generation maintains a linked list of objects associated with that generation. After scavenging of a generation is completed the linked list is traversed and unmarked big objects are deallocated. Promoted big objects are transferred to the next higher generation's BOT list.

At garbage collection time the collector will need to find all the possible references to each object. Rather than scan the entire object space on every collection a list of references from outside generations is maintained for each generation. This list is the remembered set. This list

may include all other generations but because of the large number of references from younger to older generations they are scanned and not included in the list.

> AWB 2011: All younger generations are GC'ed immediately before GC'ing an older generation. Hence, the younger generations can be linearly scanned for references to the generation being collected.

The list is a sequential table, contained in segments (usually one) linked from rememberedSegmentList. The segment which is accepting new entries is always the firs segment in the list. The entries in the list are object references to objects which may contain a reference to an object in this generation. It is "may" because such a reference may have been overwritten.

This list is updated on every store to an instance variable. Each object contains a bit field (Rem.) with one bit for each generation. If the object being stored into has the bit set which corresponds to the generation for the object which is being stored, then an object reference to the storing object is placed in the remembered set for the generation of the stored object and the bit is cleared. This gives one object reference for each cross generation object. If a generation is expected to have a large number of such references then the bit is initialized as reset and that generation is completely scanned when the generation of the stored object is collected. (In this system this done for all the younger generations.

> AWB 2011: They don't occur in Java (at least circa 1998) but heap allocated activation records or closure environment records would probably be treated this way so stores to heap allocated local variables wouldn't need write barriers. This was done in the Smalltalk implementations that were precursors to the JOVE GC design.

References from shared objects to thread local objects are not placed in the remembered set. If a reference to a thread local object is placed into a shared object then the local object is potentially share able. At this point the local object and it's referents are transitively copied to global space. Then the local space is scanned and all references to the copied objects are updated. This scan may be shortened by keeping track of the generations from which objects were copied, and then using remembered sets to find references from other generations. At the end of this copy operation the shared spaces will need to be checked to see if they should be scheduled for garbage collection.

If we determine that it is worthwhile to stack allocate objects then a separate generation (-1) will be added. Objects will be allocated in this generation as normal, although there will be a separate IBIC instruction to force an object into the generation. Large objects and arrays will never be allocated in this space. This space will never be garbage collected. Since objects in this space will never be stored into other objects it needs no remembered set bit.Any method which may stack allocate an object must call a run time routine which will return a "magic cookie" which must be stored as a binary value within the routine. This call must appear before the first stack allocation. Before the method returns but after the last reference to any stack allocated object another runtime routine must be called with the "magic cookie" as a parameter. This routine will free any stack objects allocated between the call that created the cookie and the call that takes it as an argument.

> AWB 2011: This scheme was never implemented for JOVE. It is essentially a Mark/Release memory management policy. The intention was that the JOVE compiler would use escape analysis to detect objects that could be managed in this manner. "IBIC" was the name of the SSA-based intermediate representation used by the JOVE compiler.

The generation table entry contains the list head for all objects which will need finalization. This list contains all objects in this generation which must be checked after each collection to see if they are still reachable. Objects which are otherwise unreachable are removed from this list, copied then queued for finalization.

There are some objects which are generated statically by the compiled code. These objects will go in their own spaces which will never be collected. The may be distinguished by those which are smaller than a segment and those which are larger. The small objects should be grouped into segments and placed in blocks which are of segment size aligned on a segment boundary. The first three words of the segment should be zero and the fourth word should be <TBD>. Each of these objects should have their header flags set to 0x02 and their remembered flags set to 0xFF. Objects which may be written to should be segregated into segments away from objects which may not be modified. Note that objects for which a monitor may be set are considered to be write able.

Large objects, larger than a segment, should be stored separately. These objects have a three word prefix of which the first and third are zero and the second word is <TBD>. These objects should have their remembered flags set to 0xFF and their header flags set to 0x06. Again objects which may be written to should be segregated from those which are immutable.

AWB 2011: This document doesn't describe the actual write barrier. Pseudo code for the write barrier when storing value v into object p is:

1. If v is an immediate value (only null, for Java), goto store.
2. If p.header.large, goto large object case
3. Mask v to get its segment base addr, Retrive the segment number as s.
4. Bit test and set p.header. remembered[s]
5. If bit had previous been clear, goto add p to remembered set s.
6. Store: p[desired field offset] := v;

Steps 1 and 2 can be statically eliminated in some cases. Steps 3-5 can be done in about 5 x86 instructions.

Message Sending

There are four byte codes in the Java Virtual Machine that invoke other methods. Of these one does not have a receiver and another has the class where look up is to occur as a constant. These may be folded into a direct subroutine call. The other two byte codes require message look up. They invoke either instance messages or messages defined in an interface. These both require look up to determine the proper method to be invoked. This document specifies that the look up for instance messages is different from the look up for interface messages. Later, if we are able, we may be able to use the interface message look up for all messages.

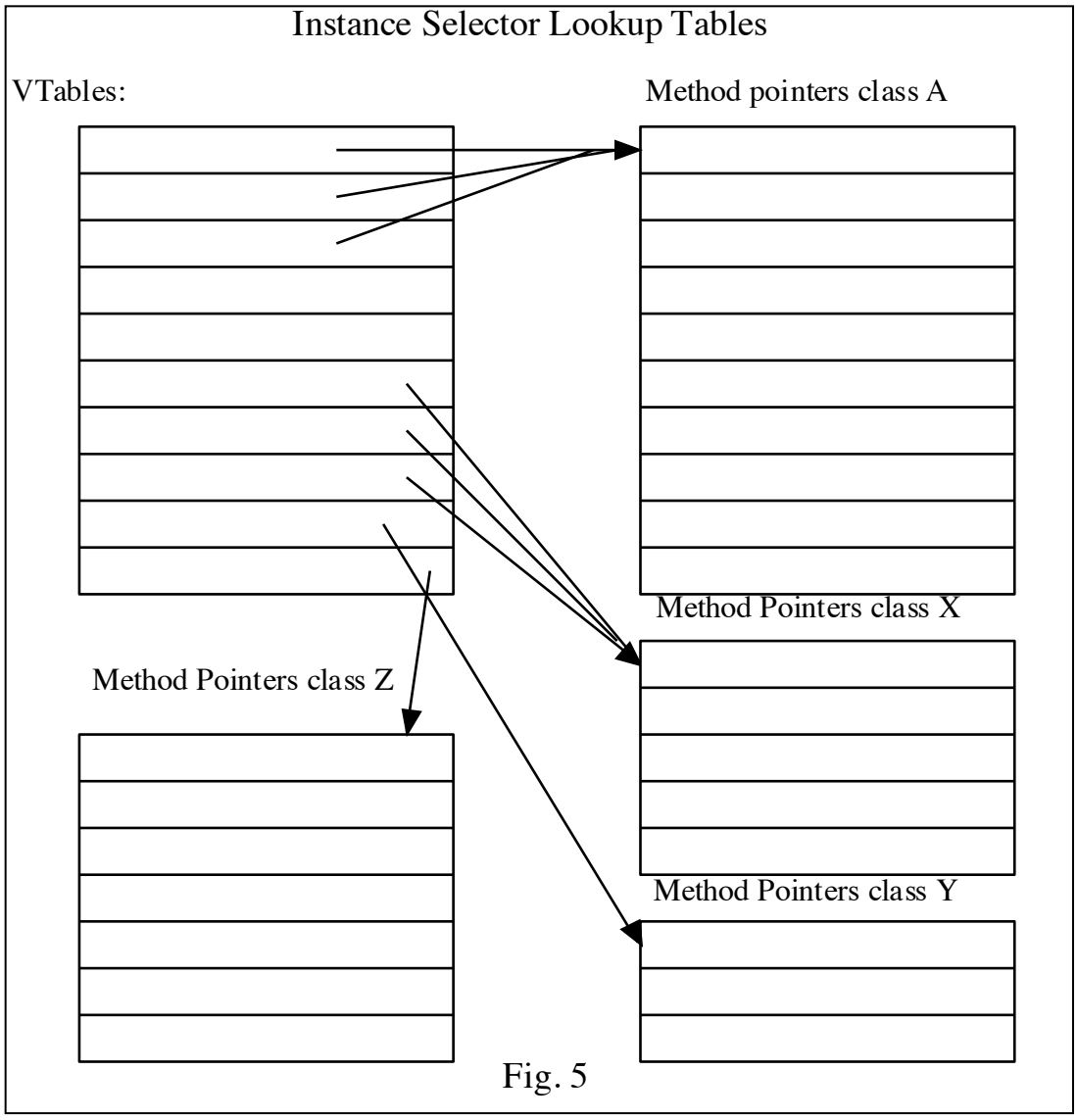

Fig. 5

AWB 2011: JOVE was actually implemented to use the "interface" technique in all cases. However that is probably a misnomer, as the dispatch technique used doesn't actually involve any knowledge of interfaces. Conceptually each selector (virtual polymorphic method) has a table of concrete implementations that is indexed by class ID. The "magic" is in how the class ID's are assigned in conjunction with logically overlapping dispatch tables. The final table sizes are very close to the collective size of normal vtables (for the same program) but all method dispatches are resolved with a single table access.

There are several optimizations that may be made in message look up. Instance messages that are implemented in the class of the type of the receiver or one of it's superclasses and not by any of the receiver type's subclasses may be constant folded into a direct subroutine call. All messages that have only one implementing method may be constant folded into a direct subroutine call. There may be other special cases. Messages where all sends are optimized out do not have to appear in the look up tables.

The tables for use by instance message sending are pictured in Fig. 5. all these tables are built at compile time and the address of any of them may be considered a compile time constant. The

collection of tables is indexed by the table VTables. This is a table indexed by the class of the receiver object. This table points to the subtable for each class. The table for a class consist of pointers to the methods to implement the various selectors. The tables are organized such that the methods for selectors implemented by the superclasses of any class appear first followed by the selectors implemented for this class. This is most easily done by copying the immediate super class table, replacing the pointers to methods overridden by this subclass then appending pointers to new methods implemented by this class. The selectors are identified at run time by their index into this table. Note that the same selector may have different encodings in different class hierarchies. To improve space utilization a class may share a table with any subclass which does not override any method. For interface messages the tables are arranged somewhat differently. The tables are inverted in that there is a table for each selector. These tables are indexed by the class to get a pointer to the method to be executed. This form of table is often called an inverted V-table. The inverted V-tables are shown in Fig 6.

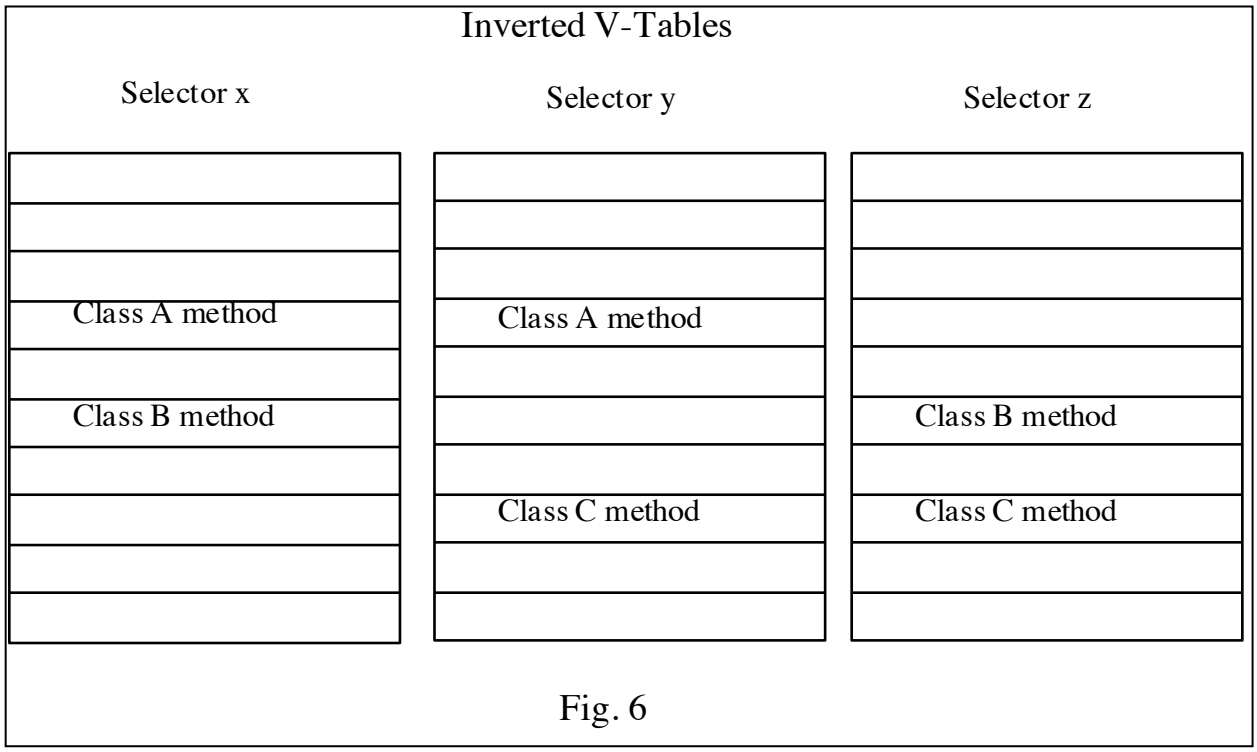

Fig. 6

One thing to notice is there are a lot of empty entries in these tables. However since the tables for the selectors are so sparse it is possible to overlap them such that the full entries in on table fill the empty entries in another. This technique is described more fully in Driesen and Holzle, "Minimizing Row Displacement Dispatch Tables" in the proceedings of Oopsla '95. The compressed tables are illustrated in Fig. 7.

Note that the subtable for different selectors is at different offsets into a master array. Also the offsets do not have to be to a point within the master table. If a selector has a only unused entries in the first part of its table then its offset may be negative with respect to the master table origin.

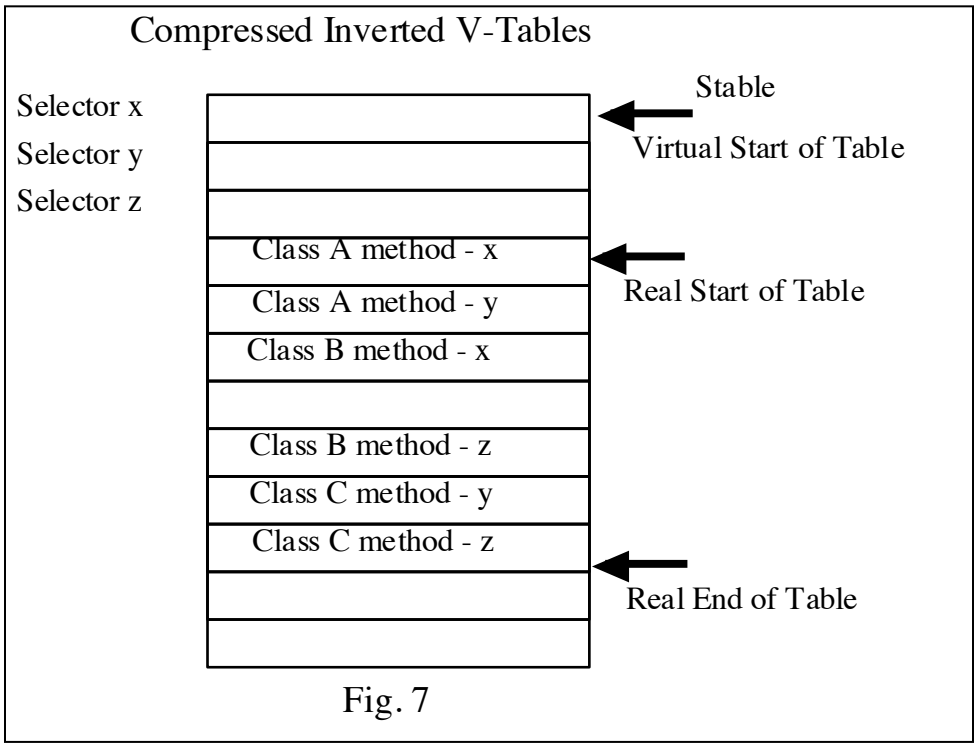

Fig. 7

The actual sending of a message using these tables may be broken into four parts. The determination of the class index, getting the address of the proper V-table, selecting the correct entry form the table, and invoking the routine. The first and last of these steps are the same regardless of whether the message is to an instance or interface selector. An example of the 80x86 code to get the class ID from the object is given below. In the following the registers are given as RA and RB. These may be mapped into any 80x86 registers but the RA in all the examples in this section is assumed to be the same register. Likewise with RB. the first example is for a system with 16 bit class ID's.

```
        move.l      RA, receiver    ; optional
        xor.l       RB, RB          ; optional
        move.w      RB, 2[RA]
```

The first two instruction may not be needed if the correct values are already in a register. The code generator may be able to test for these cases. If the class ID is expanded to 17 bits then the following code will need to be generated.

```
        move.l      RA, receiver    ;  optional
        xor.l       RB, RB          ;  optional
        move.w      RB, 2[RA]
        test.w      [RA], #0x4000
        beq.s       $1
        add.l       RB, #0x10000
$1:
```

This code must be generated if the type of the receiver or any of it's subclasses has a class ID that will not fit in a 16 bit integer. If the type and all of it's subclasses have 17 bit class ID's then the test and branch instruction may be deleted. Removing the branch will be extremely helpful because of piplining considerations.

> AWB 2011: We considered the possibility that we would need class ID with more than 16-bits. We didn't implement it and never encountered a program that needed more that 16-bits, although we did encounter some that used more than 15-bits.

Getting the address of the V-table is then next step. This should only generate executable code for instance methods. For interface methods the table depends on the selector which is known at compile time and may be constant folded. the register RB is the same as the RB in the sample code to get the class ID. This is separated from the get class ID for two reasons. one, getting the class ID may be a common sub expression with other byte codes which do such things as casts. Secondly, there is some advantage to having several instructions between this and the get class ID to allow data to pass through the machines pipelines.

```
move.l      RA, vtables[RB*4]
```

Next the actual method is fetched from the table. this is a separate operation from finding the V-table since that operation may be a common sub expression for several messages sent to the same object. There are two forms of this first is the form for a send to an instance selector. RA is the same register RA as the previous example. The value methodIndex is constant assigned to the selector and is the index into the V-table for this method. It is constant over the implementing class and all of it's subclasses including those that override the message. It may be a different constant in class hierarchies that do not have a common implementation of a method with this selector.

```
move.l      RA, methodIndex[RA*4]
```

For interface methods the code is:

```
move.l      RA, stable+seloff[RB*4]
```

The registers RA and RB are the same registers as the previous examples. Stable is the, perhaps virtual, start of the condensed inverted V-table and seloff is the offset to the start of the subtable for this selector. Both of these constants are assigned by the code generator.

A method is invoked by the following instruction sequence. The register RA is the same as the proceeding code fragments.

```
call        [RA]
```

Threads

There is a fairly straight forward mapping of Java threads onto the threads in Win-32. There is additional data needed by our implementation. This data will be contained in a additional data structure allocated when the thread is started. A pointer to this data will be stored in the

privateInfo in the existing Thread class. The data structure will contain the windows thread handle, the Thread ID, the memory space descriptors and other information needed about this thread. This data structure will be created when the run message is sent to the thread. At this time an OS thread will be created and started. A pointer to this structure will be stored in Thread local storage and will be accessable with the system class TlsGetValue. These actions will be reversed when the thread is exited. A thread storage object will be staticly allocated and used for shared data activities. Ths structure will only be partially populated. A Thread object will never be garbage collected while it is active. The only other question is the mapping of the Java execution priorities to the priority class and sub priority used by windows. The following table gives the proposed mapping.

Java	Win Prio class	Win priority
1	IDLE_CLASS	IDLE
2	IDLE_CLASS	BELOW_NORMAL
3	IDLE_CLASS	ABOVE_NORMAL
4	NORMAL_CLASS	LOWEST
5	NORMAL_CLASS	NORMAL
6	HIGH_CLASS	LOWEST
7	HIGH_CLASS	NORMAL
8	HIGH_CLASS	HIGHEST
9	REALTIME_CLASS	NORMAL
10	REALTIME_CLASS	HIGHEST

Monitors

The assumption is that there will be few objects with active monitors. However alternative implementations are proposed if this assumption is found to be incorrect. Since it is assumed that there will be few objects with a monitor only one bit in each object will be allocated to indicate there is monitor information available. The information required by a monitor will be stored in a separate table and found by searching a table of active monitors using the object address as a key. The first implementation will use one table to hold the monitor information. It will be a simple array sorted by object address and objects searched for using a binary search. If this proves too inefficient then the table will be split with a different table for each grade space. If this is still too inefficient then the tables will be converted to hash tables (one per grade space).

Since a Java program does not share variables across processes the Win-32 construct CriticalSection will be used as the underlying mechanism. When the program first syncronizes on an object, which can be determined by looking a the monitor bit in the object header, a CriticalSection will be created and the handle will be stored in the monitors table using the object address as a key. If the monitor has previously been created the monitor handle is retreived from the monitors table. The system will then do an OS EnterCriticalSection on the monitor handle.

After a grade space is garbage collected the table is searched and if any objects have been moved then the table key is updated. If an object is dead then the monitor is released and the entry is deleted from the table.

In order that the monitor will be cleared during the stack unwind that occurs during exception processing the compiler should generate a finally block to unlock the monitor. The try block should include all the code from the monitor lock to the monitor exit.

Exceptions

A method can be thought of as being broken into regions which are protected by exception handlers or finally handlers. These regions may be properly nested. When such a region is entered the executable code activates the desired handlers, and on exit from the region it restores the handlers for the containing region. The method itself may be considered to have the null exception environment, although synchronized methods may have their entire executable wrapped in a finally region to release the monitor for the method.

There is a thread global pointer to the topmost stack frame with an exception environment. Within the stack frame there is a variable (Link) which has the address of the stack frame for the next previous stack frame with an exception handler. These links point to the return address element of the stack frame. These pointers form a linked list of stack frames with exception handlers.

Methods with exceptions or finally blocks have several variables, besides the Link, in their stack frame to control their exception environment. Methods which do not have any exception handlers or finally blocks do not need these variables on the stack, and do not have to be linked into the list. However for debugging a compiler option may force an exception handler for every method. The variables are the AXE and Name variables.

The Name variable is optional, it will only appear if the debugging option is enabled. It will allow the system to provide meaningful information from the Throwable.printStackTrace message. It is constant during the execution of a method and is a pointer to name information for the method. This value is used to generate debugging stack traces. If the name variable is not included in the stack frame the stack trace will show the return address instead.

The AXE variable is a pointer to the active execution environment block. At method entry it is set to point to a default XEB. This is usually the null exception block which handles no exceptions or finallys, which is signaled by having a null pointer (zero) in this variable. However for synchronized methods it should initially be set to a compiler generated XEB which handles no exceptions but has a finally block defined to release the monitor for this method. On entry to/and exit from a region this variable is adjusted to point to the proper handlers for the region/for the containing region.

An exception environment block XEB defines the exceptions to be handled, and the order that they are to be tested, for a given region of code. The address of the currently active one is is stored in the AXE variable in the stack frame. The first half word (16 bits) of the XEB is an unsigned count of the number of exceptions that are defined by this exception handler. The next half word is the number of bytes that are expected to be pushed onto the call stack for the current stack frame when an exception handler or the finally handler for the current region is invoked. This word is used to discard work in progress on the stack by trimming the stack back to the number of bytes specified when a handler is invoked. This value is the number of bytes on the stack including stack frame overhead. It will probably to be the same for all exception blocks in a given method. The next word is a pointer to the code which will handle the finally execution.

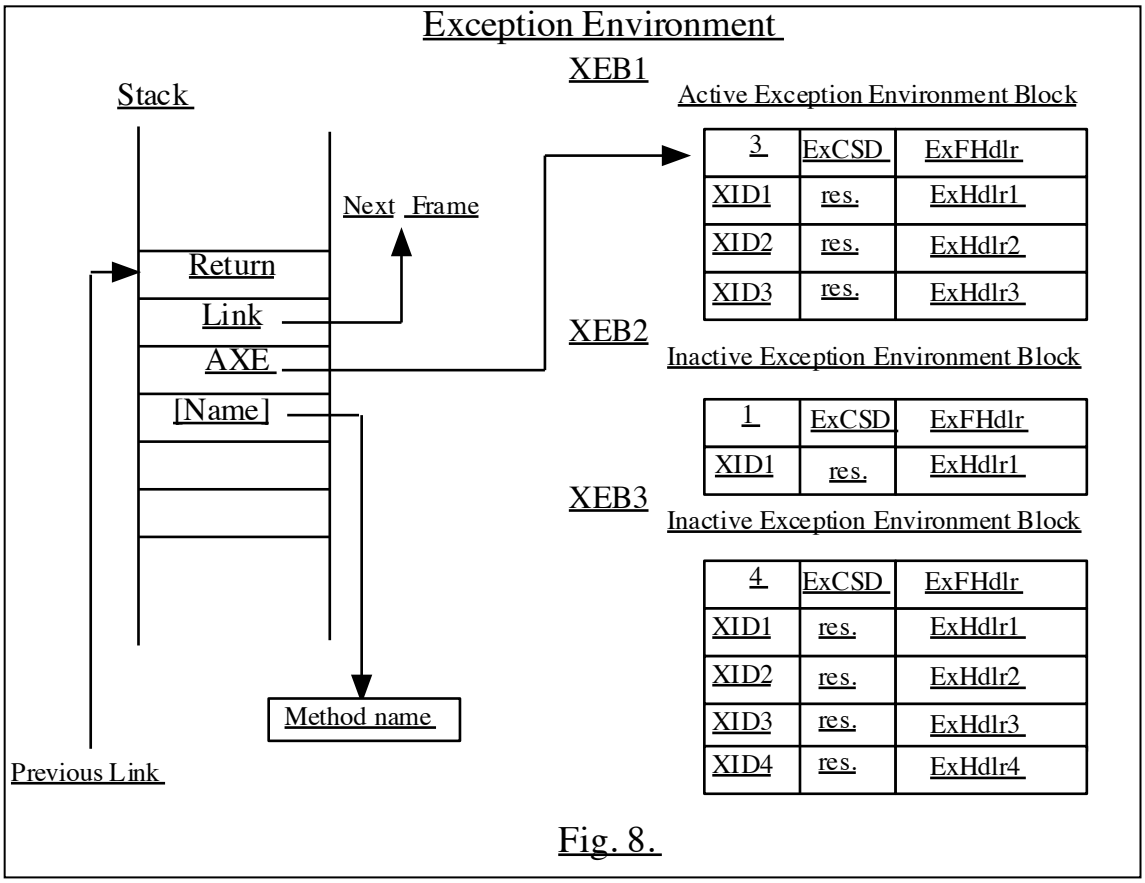

Fig. 8.

The rest of the XEB consists of as many two word entries as was specified in the count field. The first word of each entry is broken into two 16 bit sub-fields. These give the lower and upper bounds for the class ID's of the exception class that was specified for this exception handler. The first half word is the lower bound and the second half word is the upper bound. The class ID's include the class ID of the exception class and all of it's subclasses. This encoding implies that the exception class ID's for a class and its subclasses must be assigned in a contiguous block. This will occur if the class ID's for these classes are assigned in a pre or post tree walk of the class hierarchy starting at class Throwable. The second word of each entry is the address of the handler code for this class of exception.

The run time environment will provide a routine to perform the throwing of an exception. This routine will accept one parameter in the ECX register which is the Throwable object. It stores the Throwable object in a thread global variable to indicate there is an exception in progress. Then it gets a pointer to the topmost stack frame which has an exception handler from the thread local memory. The compiled code is responsible for maintaining this pointer. The routine will never return to the calling point so all other registers including the c-stack pointer are volatile. It will process the exception by taking the class ID of it's parameter and, starting at the top most stack frame, find the entry in the active XEB which has that class ID within its range of exceptions. If it finds such an entry it will first trim the run time stack to the depth indicated by the XEB ExCSD entry, load the Throwable object into the EAX register and place the NULLVALUE into the thread global variable and then jump to the handler code. If no entry is found the finally handler for the region is called. If the finally handler returns the lookup is restarted with the new exception environment for the current stack frame. If the AXE or finally handler is null then a new stack frame is retrieved from the Link variable.

When a region of code with an exception/finally handler is entered install the XEB for the range in the AXE variable. At the end of processing for this region if there is a finally block that covers this region then invoke the finally handler by calling it as a subroutine with a register parameter of the stack frame pointer. If there is no finally handler then copy the address of the XEB for the surrounding range into the AXE variable. A return must invoke all of the finally handlers for this and any containing regions in the method before actually returning.

A finally handler may be invoked in two ways. It may be executed when the protected code finishes normally or it may be triggered by the abrupt completion of the protected region. The generated code must invoke the finally handler at the end of the region. If the region is abruptly terminated by a return then the generated code must invoke the finally handler. If it is abruptly terminated by a throw then the exception handler will automatically execute the finally handler.

When it is invoked, a finally handler is called as a subroutine with the EAX register pointing to the stack frame (the return address for the method). The return address for the finally handler is on top of the stack. The exception run time will have trimmed the stack to the value indicated by the active XEB. If this is incorrect or if finally block is invoked from the generated code, there may be additional values pushed on the run time stack beyond those is expected by the Java code. In that case, the handler must first save the return address and then adjust the stack pointer to it's expected offset from the stack frame base in EAX. Next it must install the XEB address for the containing range in AXE. Then the Java statements may be executed. At the end the code should return to the address which was on top of the stack when the handler was invoked.

When it is invoked an exception handler will have the runtime stack trimmed to the value indicated in the XEB. The EAX register will contain the thrown object. An exception handler never returns to its invoker so there is no return address.

An exception handler for a region with no finally operates in the exception environment for the containing region. If there is a finally for this region then the environment is the finally for the current region but with no exception handlers. A new XEB is first installed in the AXE variable. Then the Java code is executed. If the exception handler completes normally then the finally handler for this region, if any, is invoked and then a jump is made to the continuation of the containing region. If the exception handler executes a return then the compiled code must invoke all of the finally handlers for this and any containing regions before actually returning.

The exception handling for native methods invoked through the JNI interface is done differently. These methods poll for an exception after each call to a Java method or to a runtime system routine.

When a native method invokes a Java method, through one of the CallxxxMethod interface routines, the interface routine sets up a stack frame which defines an exception environment. This environment only contains a finally entry. The finally code will perform a return to the native method. The value returned will be zero for integer or Boolean types and NaN for Float and Double returns. The exception will be left pending and may be checked by the native method. If the native method returns without clearing the exception, the compiled code will check the exception status and rethrow the exception.

Asyncronous exceptions will normally be treated as syncronous the exception will be thrown at the time that it occurs. However there will be a thread local variable which indicates that the thread is in a region where this is invalid. In such case the exception will be stored in the current exception variable and left pending. When the compiled code leaves the region where exceptions are invalid it must check the current exception and throw the exception at that time.

Stack Frame

There are three different forms of stack frame. The normal or short frame is for methods with no **try** statements. The long frame for methods which do have **try** statements. Long and short frames may be intermingled on the stack. A debugging stack frame is only generated when specified by a compiler option and in that case it is generated for every method. This stack frame is similar to the long frame but contains additional information used to generate call trace information during exception handling.

The stack is divided into two separate stack segments which run in parallel. One stack, called the c-stack (call stack), contains all the binary data. The other stack contains all the object pointers which would normally be stored in the stack frame. This separation allows a precise garbage collector to update the object pointers without confusion with binary data Fig. 9 shows the short stack frames for two methods (M1 and M2) as they would be arranged on the stack. The top of the C-Stack is is pointed to by the ESP register. There are no explicit links from one frame to its predecessor..

> AWB 2011: We had used this duel stack approach in Smalltalk implementations and found that it also worked very well for Java. It simplified GC because the collector it doesn't have to "parse" the stack to find object references. The GC just uses the entire current extent of the object stack as its dynamic roots. A further simplification is that all native calls are via JNI and individual object stack entries can be used as JNI local handles.

The OSB (Object Stack Base) is a pointer to the bottom of the object stack frame for the method. It is pushed by the invoker and points to the last slot used by the calling routine. This object stack element is the first argument to be placed on the object stack by the callee if necessary. Otherwise it is the last element used as a temporary by the caller. References to values on the object stack are relative to this value. The arguments to the method including the receiver are then loaded into registers or pushed onto the appropriate stack by the invoking routine. The diagrams show only one such argument on each stack but there may be any number from none to 65535. The return value is the memory address for a return to the invoker. It is pushed by doing a normal CALL to the entry of the method.

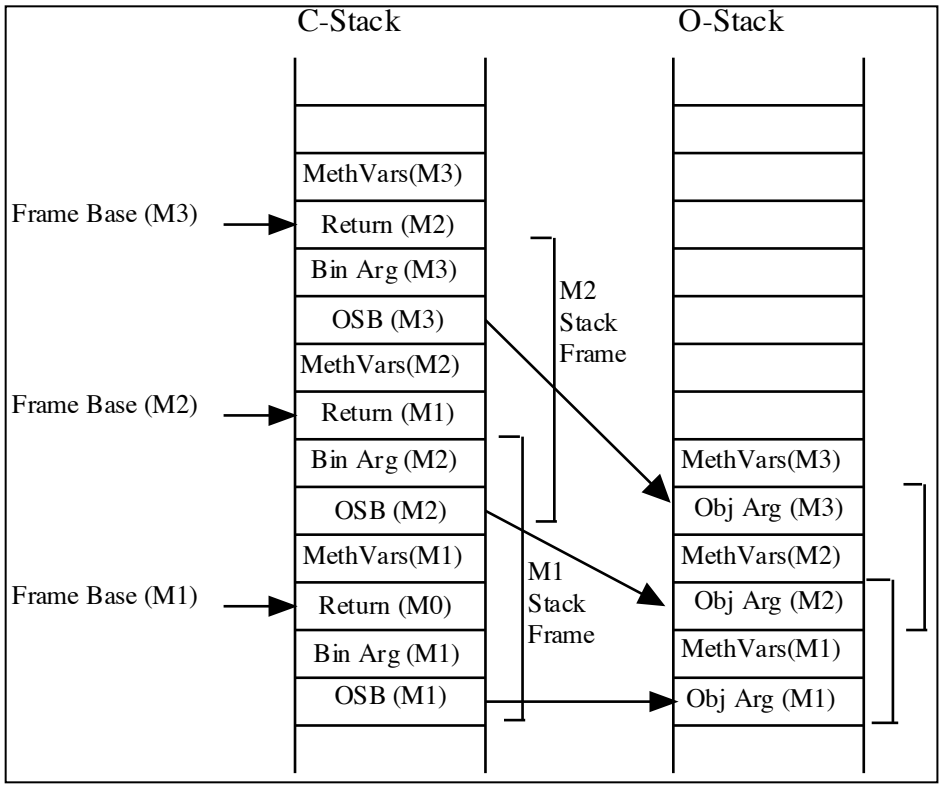

Fig. 9.

The invoked routine must first allocate the correct amount of memory for both binary and object method local variables. The diagrams show only one such value on each stack but there may be any number including none. These values will be initialized, the binary values to zero and the object values to NULLVALUE. The first argument is passed in the EAX register if it is not a long, float or double.The second argument, if there is one, is passed in the EDX register unless it is a long, float or double. A long is always passed as a pair of words in the binary arguments section of the stack frame. The first float or double argument is passed on the hardware floating point stack. If there are two float or double arguments then the second is also passed on the hardware floating point stack. The first argument will appear below the second in this case. For instance methods the first argument is always the receiver and is always an object reference in EAX.

A long stack frame appears in Fig. 10. The frame is quite similar to the short frame but includes several additional variables in the fixed part of the frame. These variables have been previously discussed in the section “Exceptions”. Note that the back link is the memory address of the stack frame for the previous method with an exception or finally handler. This may skip one or more methods which have only short stack frames.

A debugging stack frame is identical to the long frame except there is one additional constant value added to the stack frame. This is a pointer to a null terminated string which contains the fully qualified name of the method that was invoked. This value appears immediately above the AXE value on the stack (between the AXE value and the first method local variable.)

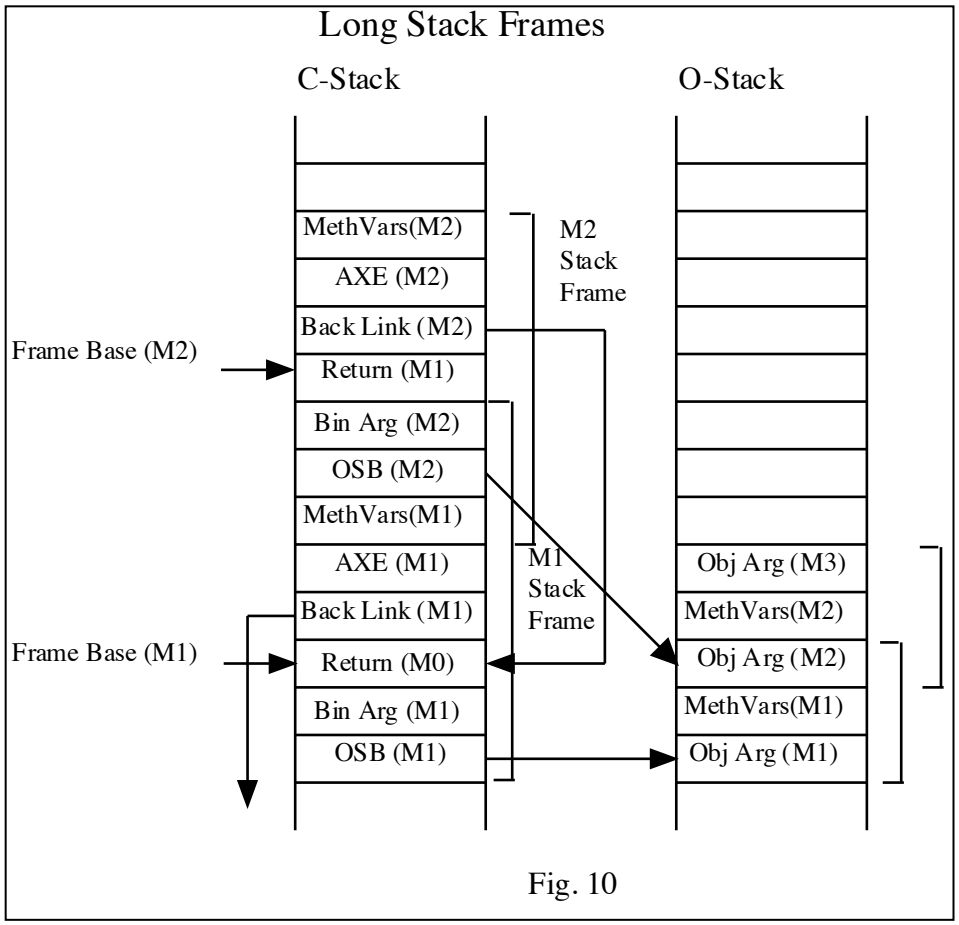

Fig. 10

The general rules for who generates and who frees values on these stacks are:

Value	Created by	Freed by
Return	caller	callee
Exception Vars.	callee	callee
Object Args	caller	caller
Binary Args	caller	caller
Object Temps	callee	caller
Binary Temps	callee	callee

Example generated code for:

call method:

```
        move.l  R0, OSBO[ESP}      ; -
        lea.l   R0, N[R0]          ;   - Set up receiver and
        push    R0                 ; -
        move.l  EAX, this          ;   - OSB on stacks
        .                          ; -
        .                          ;   - code to load/push arguments
        .                          ; -
        call    [RR]               ; Call method
        add.l   EAX, #M            ; remove arguments
```

Method prolog (short stack frame)

```
sub.l     ESP, T          ; space for binary temps
.                         ; Code may be needed to
.                         ; store EAX and EDX.
```

Method prolog (long stack frame)

```
sub.l     ESP, T          ; space for binary temps
xor.l     R0, R0          ; Get a Zero
move.l    AFH[ESP}, R0    ; null finally handler
move.l    AXE[ESP}, R0    ; null exception handler
move.l    R0, EXPTOP      ; Get current exception handler
move.l    LINK[ESP}, R0   ; Set back link
lea.l     R0, LINK[ESP]   ; Get current link
move.l    EXPTOP, R0      ; Set global with top.
.                         ; Code may be needed to
.                         ; store EAX and EDX.
```

Method Return

```
add.l     ESP, T          ; remove temporarys
ret
```

Where

M	is the number of binary method arguments for called method times four.
N	is the number of object arguments and method temps for current method times four.
OSBO	is current offset from stack pointer to OSB.
R0	is an arbitrary free register.
R1	is an arbitrary free register.
RR	is the previously resolved method address register.
T	is the number of binary method temporaries for current method times four.
U	is the number of object method temporaries for current method times four.

Register Conventions

Due to the few registers available in the 80x86 and the desire to use an efficient register allocation scheme, the ESP register is dedicated register at run time. Other registers may have

assigned tasks on entry or exit from a subroutine but they are all otherwise available to the code generator. No registers are saved across a subroutine call however several may have defined uses on entry or exit. The values on the floating point stack are not maintained across a subroutine invocation but the stack may be used to pass or return values. No object references may be left in a register at any point where a garbage collection any occur.

EAX	On entry to a subroutine it contains the first argument or nothing On exit it contains the return value or the LSW of a long return value.
EBP	Scratch register - will not be used for addressing objects.
EBX	Scratch register
ECX	Scratch register
EDI	Scratch register
EDX	On entry to a subroutine it contains the second argument or nothing On exit it contains the MSW of a long result otherwise nothing
ESI	Scratch register
ESP	Always points to the last used value on the C-Stack.

The floating point stack may contain one or two floating point values on entry to a subroutine. It may contain one value at exit. All other elements of the stack are volatile across a subroutine call.