

# The Tiger Virtual Machine and Runtime Environment Specification

Chris Lattner  
September 28, 1999  
CS490: Advanced Compiler Design  
University of Portland

The Tiger Virtual Machine and Runtime Environment.....	1
Target Audience.....	1
Tiger Compiler Overview.....	1
Tiger VM Overview.....	1
Tiger VM Goals.....	1
Tiger Runtime Environment.....	2
Tiger Variable Types.....	2
Tiger Memory Subsystem.....	2
Heap Object Structure.....	3
Raw Bit Settings:.....	3
Forwarding Bit Settings:.....	3
Garbage Collector.....	4
Finding Roots for the Garbage Collector.....	4
Stack Scanning Algorithm Constraints.....	4
Stack Scanning Algorithm #1.....	5
Stack Scanning Algorithm #2.....	5
Object Referencing Semantics.....	5
Tiger Runtime Conventions.....	5
Tiger Function Call Semantics.....	5
Register Save Conventions.....	6
Summary.....	6

## The Tiger Virtual Machine and Runtime Environment

The Tiger VM is the backend interface from the “IR With an Interesting Name” (IRWIN) to the x86 host architecture. It is defined to do the most host-specific optimizations, and eventually code generation for the Intel 80386 processor running the Linux operating system. This paper describes the Tiger runtime environment, the Tiger VM, and the IRWIN to VM translator module.

### ***Target Audience***

This paper describes a VM implementation for the language Tiger. “Tiger” is a high level programming language described by Andrew Appel in the book titled “Modern Compiler Implementation in Java.” Although no knowledge of Tiger is required for this paper, it would be helpful in general to have.

Because we will be describing in depth the translator module, a solid understanding of the Intel 80386 architecture is crucial, as is the IRWIN architecture. Please refer to the IRWIN design document for more information.

### ***Tiger Compiler Overview***

The Tiger compiler is made up of several stages that build three disjoint data structures. The first data structure built is the Abstract Syntax Tree (AST), which is created by the Parser phase of the compiler. The second data structure is the Intermediate Representation (IR) which is created by the AST to IR translator stage. Last, the IR is translated into the VM data structure. This process is used to facilitate multiple levels of optimization and program analysis, which allows the Tiger compiler to generate high quality code.

### ***Tiger VM Overview***

The Tiger VM is an 80386 (and compatible) specific representation of the Tiger IRWIN, with the appropriate runtime support library. This representation is chosen to allow 80386 specific optimizations to take place, such as instruction selection, register allocation, and peephole optimization in an efficient and elegant way. Once the VM data structures have been built, and the optimizations performed, 80386 assembly language code may be emitted by the VM.

### ***Tiger VM Goals***

The main goal of the Tiger VM is to provide a platform that supports experimentation, optimization, and future enhancements. As such, it must be general enough to allow adaptation to the future, yet powerful enough to yield an efficient system that may be highly optimized. In addition, several constraints have been imposed on the implementation of the Tiger VM:

1. Tiger is a garbage-collected language. As such, the virtual machine must provide a execution environment that is compatible with the runtime garbage collector.
2. Interfacing with C code. The Tiger language will be enhanced to allow support functions to be written in C code, seamlessly integrated with functions written in Tiger. To support this, a Tiger language extension will be defined to allow the

definition of “external” C functions. To support this feature, we must be compatible with C calling conventions, as well as provide a mapping between Tiger and C data types. Additionally, C programmers must be careful to follow the constraints needed to work in a garbage collected environment.

3. The x86 architecture. Unfortunately, the target architecture is not a very orthogonal architecture, which makes several tasks difficult. Optimization becomes a complex combination of instruction scheduling, pipelining, and register allocation. Register allocation, however, is further constrained by the fact that certain primitive operations (ex: `mul/div`) only work with certain registers (ex: `[e]ax/[e]dx`).

These constraints must be satisfied for the Tiger compiler to work at all, although they need not be implemented in the most efficient way the first time. It is planned that the Tiger compiler will be iteratively improved over the course of the semester. This document only describes the initial implementation.

## ***Tiger Runtime Environment***

In order to make much sense, the Tiger VM must be considered in light of the runtime environment set up. As mentioned, this runtime environment is designed to be garbage collector compatible, allowing all “roots” of the object references to be easily detected.

## **Tiger Variable Types**

Variables in the Tiger runtime may only be of two types, both of which are 32 bits in size. Integer values (declared as type ‘`int`’ in Tiger), are stored in a special intermediate form – shifted left by one bit, with the low order bit set to zero. This gives integers in the Tiger runtime a range of 0 to  $2^{31}$ , and slightly complicates multiplication and division operations.

The other variable type is a Heap Object Reference. Heap object references are simply pointers into one of the Tiger object heaps (see below), with the low bit set to 1 (value stored is `<real object ptr>+1`). Having the low order bit set to one allows the garbage collector to easily distinguish between heap object references and integers.

Strings are represented as a ‘raw’ heap object, containing eight-bit binary string data. No terminating null character is stored in the string data, instead the string length is stored in the garbage collector header word. The value of NIL in Tiger is stored as the integer 0, allowing it to be treated as an integer, not an object reference.

## **Tiger Memory Subsystem**

The Tiger memory subsystem is broken into several distinct sections of memory. At compile time, the compiler is allowed to allocate constants and global objects in the Statically Allocated Heap (SAH), which is not garbage collected. At run time, the client program dynamically allocates memory from the Garbage Collected Heap (GCH). Additionally, a stack segment is statically allocated by the compiler, and a `malloc()` heap is provided for the internal use of the C runtime code (not to be exposed to Tiger).

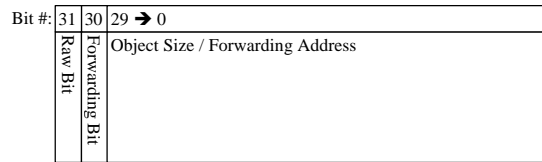
The SAH is implemented by using the ‘`.data`’ section of the executable to pre-initialize constants and global objects. The GCH is implemented by using two equally sized `mmap`’ed regions of memory (two regions are needed because a copying garbage collector will be used, see below). The standard UNIX stack segment will be used for the Tiger stack, and the standard `malloc()` arena will be used for the Tiger `malloc()` arena support.

## Heap Object Structure

To facilitate efficient garbage collection, all Heap Objects have a uniform structure. The garbage collector is aware of two distinct types of heap objects: raw and marked. The Tiger compiler, however, maps several distinct types (Records, Arrays, Tiger Objects, Strings, and Dispatch tables) onto these two GC types. Type-safe access is provided by the type-checking phase of the Tiger compiler.

In addition to the heap object type, the garbage collector has to keep track of some per-object bookkeeping for when doing the actual collection operation. To maintain all of this information, the garbage collector prepends a 32-bit header to every block of memory allocated. This header contains a bit for the object type, a “forwarding” bit, and the object size.

This memory block header is laid out in memory with a structure shown on the right. The most significant bit (MSb) contains the “raw” bit, which is set as follows:



Heap Object Header

### Raw Bit Settings:

0. When set to ‘0’, the object is treated as a “marked” object. A “marked” object is made up of 32 bit chunks of data, which is treated as possible heap references. By annotating a block as “marked”, the VM ensures that all 32 bit words can be determined to be either object references or not (see Tiger Variable Types, page #2).
1. When set to ‘1’, the object is treated as a “raw” object. This means that the bits of the object are never interpreted as anything other than raw data.

The forwarding bit also used internally to the garbage collector. This bit is used when actually in the process of a garbage collection phase. If an object has already been forwarded to the new heap, this bit is set, with the following effects:

### Forwarding Bit Settings:

0. When set to ‘0’, either garbage collection is NOT occurring, or the object has not been forwarded yet. When this bit is cleared, the Object Size / Forwarding Address field of the Heap Object Header contains the size of the object in bytes.
1. When set to ‘1’, a garbage collection process is underway, and this object has already been forwarded to the future heap. If this is the case, then the Object Size / Forwarding Address field contains the absolute address of the new object’s location, shifted to the right by 2 bits. This ensures that objects may be placed anywhere within the  $2^{32}$  bytes of addressability.

The forwarding bit is used to control the interpretation of the Object Size / Forwarding Address field of the header. When the Forwarding Bit is cleared, the Object Size / Forwarding Address field contains the “real” size of the object (not including the Heap Object Header). It is possible that this size is not a multiple of four (for example strings), and is not shifted to get the real value of the field (as the Forwarding Address field is). To get the allocated size of the object, the garbage collector uses the formula  $((\text{stored size} + 3) \& \sim 3)$ .

## Garbage Collector

When a dynamic heap allocation is performed, but all available space is exhausted, a garbage collection pass is started. A copying garbage collector was determined to be the simplest efficient garbage collector that we can implement, given the time-span of this project. The garbage collector will use the runtime stack and the SAH to locate the roots of the object reference tree. From that, it will continue scanning into the allocated object heap, copying as necessary. It is out of the scope of this paper to completely describe the garbage collection algorithm.

### Finding Roots for the Garbage Collector

A copying garbage collector operates by starting with a set of “root” object references, and tracing down the objects that they reference, the objects that the references objects reference, etc. until there are no objects left with outstanding references. The most crucial part of this delicate process is finding the root object references.

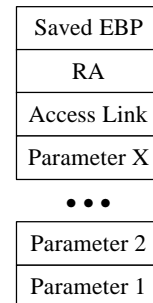
The Tiger x86 VM finds root references in two places: global variables and the stack. Global variables are easy to check for references, because they all live in the SAH, which does not move, nor change during the garbage collection process. The low bit of each word of the SAH is checked. If the bit is set (see Tiger Variable Types, page #2), the word of memory is treated as an object reference, and the object it refers to is marked as “alive”.

The Tiger x86 VM does not need to check registers for roots, because all registers get spilled onto the stack when the “`alloc`” instruction is executed. These values are not allowed to point into the middle of object (see Object Referencing Semantics, page #5), so all values on the stack should be stored in their correct marked form.

### Stack Scanning Algorithm Constraints

Unfortunately, when it is time to scan the stack, things become more complex. The Garbage Collector must know which words on the stack are object references and which are data to be disregarded. For the initial implementation, we simplify the problem by imposing the following restrictions:

1. Native C functions may not invoke the garbage collector. This implies that they cannot invoke the “`alloc`” IR instruction, nor allocate garbage collectable memory in other ways. This limitation will be eliminated in the future, because the runtime string library must be able to allocate memory.
2. Native C code is not allowed to call back to Tiger code. This is because the C functions could cause arbitrary non-marked data to be pushed onto the stack. This is unlikely to be changed in the future.
3. Every stack frame must have a saved EBP register, and it shall be the first value pushed onto the stack upon function entry.
4. No “raw” data may be pushed onto the stack, only data stored in the “marked” form.
5. The access link is treated as the  $x+1^{\text{th}}$  parameter to the function. Since it points to an aligned stack address, it appears to be a marked integer, not affecting object references. Similarly, the EBP chain is also word-aligned.



Example Stack Frame

With these constraints, the only 'raw' data on the stack will be the return addresses (RA in diagram) which return to an arbitrarily aligned 80386 instruction. There are two different approaches to handling this situation, but at least to begin with, only the first will be implemented.

### Stack Scanning Algorithm #1

We know the top and the bottom of each stack frame, from the ability to walk to EBP chain. With stack scanning algorithm #1, all data on the stack is considered potential marked data (which may contain object references), except the word of memory that has a physical address of four bytes above a saved EBP. This word of memory is required to contain the return address for the function by restriction #3, above.

### Stack Scanning Algorithm #2

It is possible instead to treat the stack as one contiguous chunk of data. Every word is inspected, and any word the low bit set is considered for further inspection. Every word of memory that reaches this point is classified by which region of memory it points into. If it references the SAH, then nothing is done. If it references the GCH, it is treated as an object reference, and followed for recursive 'aliveness' checking. If the word references neither memory segment (a return address would point into the code segment), then it is treated as raw data and ignored.

### Object Referencing Semantics

Garbage Collection may happen any time that the "alloc" IR instruction is executed. As such, the VM must make sure that all live heap objects have live references to them on the stack when the "alloc" IR instruction is invoked. To do so, the IRWIN representation of a program may not have values stored in temporaries that point into the middle of an object. If this occurs, the garbage collector will see the reference and treat the word it points to as an Object Header, which will lead to disastrous behavior.

### Tiger Runtime Conventions

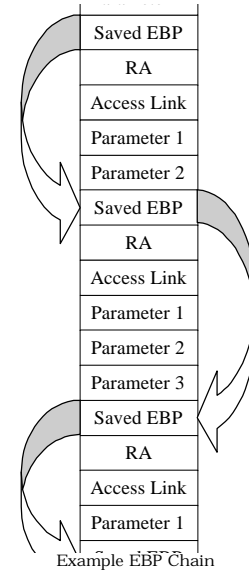
In order to maintain consistency across the implementation of the Tiger VM, a number of conventions are established, and must be followed. These conventions define how functions are called, parameters are passed, and how the garbage collector interfaces with the runtime stack.

### Tiger Function Call Semantics

Tiger functions are quite simple to implement because they have a number of restraints on them at the IR level:

- All functions take a constant known number of arguments.
- No functions have up-level variable references.
- Functions have a single defined entry point.

Tigers functions follow the convention that the caller pushes arguments and the callee pops them. Because functions have a constant number of arguments, it is a trivial matter for the callee to pop the correct number of values off the stack. Note that this is



not the case for functions implemented in C, so a different protocol will be used for calls to C functions.

When returning, a function's result value is placed in the EAX register before returning to the caller.

To implement function calls, the Tiger VM must implement the “sub”, “return” and “call” IRWIN operations. These are implemented as follows:

sub: For the example of: “sub multiply(...)”, this operation causes the generation of the following code:

```
multiply:          ; Label to define entry point
  push EBP         ; Save the old value of EBP
  mov EBP, ESP     ; Save the base pointer into EBP
  sub ESP, <i>     ; The value of <i> depends on the number of locals
```

return x: The return operation would be implemented as shown below:

```
mov EAX, <x>      ; Where <x> is a constant or a temporary
mov ESP, EBP     ; Atomically pop values off of the stack
pop EBP         ; Restore the value of EBP for caller
ret <n>          ; The value of <n> depends on the number of parameters
```

call: For the example of: “call multiply(<a>, <b>)”, this operation causes the generation of the following code:

```
push <a>         ; Pass parameter <a> in source order
push <b>         ; Pass parameter <b> in source order (probably an AL)
call multiply    ; Call the function, callee cleans up parameters
```

## Register Save Conventions

In the Tiger VM, register saving conventions follow that of the C model. That is that the callee saves the registers ESI, EDI, EBP, and EBX. The caller must save registers EAX, ECX and EDX if desired.

## Summary

The Tiger Virtual Machine is defined with an initially simple implementation that may be extended easily for the future. This will be important when we later enhance the Tiger language to support constructs that do not fall within the domain of support provided by the Tiger VM.

A clean and simple Virtual Machine will be elegant to write and easy to maintain.