Real-time Memory Management System for a Java Processor

A. Desai[1], J. Singh and R. Veljanovski
Centre for Telecommunications and Microelectronics,
Victoria University, P.O.BOX 14428, Melbourne, Victoria, Australia.
Email: [1]aniruddha.desai@research.vu.edu.au

Abstract

This paper presents a memory management system offering real-time guarantees for Java object access, along with accurately predictable memory management functions. The primary goal of the design is to allow precise worst-case execution time prediction for all memory management and object access functions. The system is designed to work with fragmented memory and offers simplified reference checking for scoped memory implementations. The presented linking model uses special runtime data structures which enable deterministic execution of object access instructions. While maintaining predictability, the system offers quick access to object data in memory; thus reducing the complexity required to implement an instruction execution unit. The results presented include time guarantees achieved for memory management and object access functions, size of the runtime image and synthesis estimates for area and speed.

Keywords: Automatic memory management, deterministic object access, real-time Java.

1 Introduction

Maintaining predictable execution times of programs is the most important objective of a real-time system designer [1]. The primary issues faced by implementation designers using Java in real-time domain are the nondeterministic behaviour [2] of automatic memory management system and the difficult to predict dynamic linking latencies. Automatic memory management creates two significant problems; the garbage collection pause times and memory allocation latencies caused by memory fragmentation. These problems need to be addressed to make java more attractive for real-time applications.

Rest of this paper is structured as follows: section 2 describes the real-time issues of automatic memory management. Section 3 presents the design of the class linker and runtime data structures. Section 4 describes the operation of object based instructions in the proposed design. Section 5 describes the proposed memory management scheme. The results obtained from the implementation are presented in section 6 and finally section 7 concludes this paper.

2 Motivation

Embedded systems have a limited amount of available memory. Frequent allocations and de-allocations create fragmentation of free and used memory. Efficiently allocating a block of memory from fragmented memory is not a trivial task, especially when the allocation times need to be small and predictable. Along with predictable allocation time, the memory management system also needs to reduce the impact of garbage collection pause time. The real-time specification for Java (RTSJ) approaches this problem by use of region based memories [3]. The RTSJ specifies a region-based allocation of objects; where each such region has a definite life time. There are many types of regions defined in the specification and the regions with limited lifetime are called scoped memories. During run-time, a reference

count is maintained for each scope that indicates the number of threads using that scope. When this count becomes zero (all computations on objects in a scope end) the region of memory representing the scope can be directly freed without using complex garbage collection and thereby offering quick memory reclamation. Though this approach does not require explicit garbage collection, it must be ensured that there is no live reference pointing to an object which was residing in a de-allocated scope. This is essential to maintain pointer safety.

To ensure pointer safety for scoped memory scheme, the RTSJ has placed a series of strict constraints on reference assignments. Table 1 from RTSJ defines all such constraints that must be followed to guarantee pointer safety [3]. While static analysis and verification can help detect possible violations, runtime checking is still required and will create a significant overhead, resulting in performance degradation.

Table 1: Reference Assignment Checks

Stored In	Reference to		
	Heap	Immortal	Scoped
Heap	YES	YES	NO
Immortal	YES	YES	NO
Scoped	YES	YES	YES*
LocalVar	YES	YES	YES

* if reference is from same or outer scope

To address these issues a new memory management model is designed, that is not affected by memory fragmentation and also simplifies reference assignment checks. Detailed description of this design is presented in Section 5. While this scheme addresses memory fragmentation and reference checking overhead, the design is developed with complete object access models, which offers a set of time guaranteed functions to create, use and execute Java's object code.

The design uses special runtime data structures that enable deterministic behaviour, even for unpredictable instructions like the 'invokeinterface' bytecode. The following section, describes the linker which is responsible for generation of Java runtime data for the memory management system. It should be noted that, although the linker performs pre-loading, it does not use static linking for dynamic structures and leaves open the possibility of dynamic class loading.

3 The Linker

The linker is implemented in Java and uses the Bytecode Engineering Library [4] from the Jakarta project. A compacted version of this software can be used with the runtime system to load classes at runtime. The linking model along with the runtime data structures produced, are the key to predictable object access at run-time. Linking model is illustrated in Figure 1.

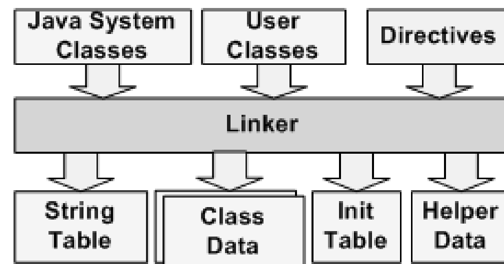

Figure 1: The Linker

The linker is also tuneable using linker directives to generate optimised runtime data according to the requirements. An example of the directives is the 'FINAL' directive which can generate a 'final' memory map of selected classes, where unused methods and fields can be stripped off to generate smallest possible runtime image. Although this will eliminate possibility of dynamic extension, it can be useful for small static applications. The linker produces runtime memory maps and supporting data structures, which are described in the following sub-sections.

3.1 String Table

For all string entries in constant pools of linked classes, a separate string table and hash table is generated by the linker. The hash table is shown in Figure 2(a) while the string table structure is shown in Figure 2(b). The choice of an 8 bit Java style UTF8 or 16 bit Unicode character representation is selectable from the linker directives.

```
hash_table{               string_table{
  32bit hash                32bit length
  32bit string*             8/16bit char[]
}                         }
```

(a) **(b)**

Figure 2: String & Hash Table

3.2 Class Data

The class data contains runtime information required by every class, to create objects, manipulate fields and execute methods. Figure 3 shows the runtime class data structure generated by the linker.

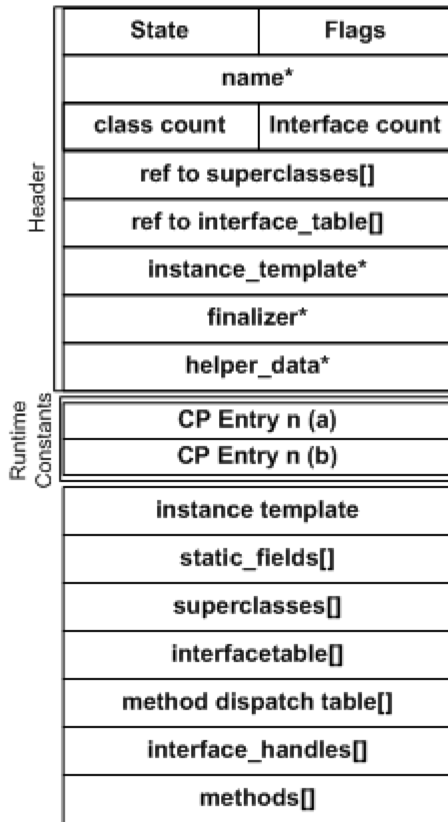

Figure 3: Class Data Structure

A fixed length header contains direct indexed information about the class. The linker starts every new class data with the three LSBs as zero, to enable quick access to eight header entires. The constant pool entries also start at a fixed location from the object pointer, so the address of any entry can be directly computed. The 'superclasses[n]' array contains one word pointer to each super class according to inheritance hierarchy (superclasses[0] will be immediate superclass and superclasses[n-1] will be 'java.lang.Object'). The interface_handles[] array, contains 2 words per implemented interface, where word '0' contains pointer to interface class data and word '1' contains pointer to an interface handle for that class. The use of interface_handles is elaborated with method invocation in Section 4.4. The instance template pointer points to an object template for this class, which helps the memory management system to create an instance of the class.

Each constant pool entry gets two words in the class data. In case of method/field reference, word 'a' is direct address or offset for methods and fields respectively, while word 'b' contains access flags and additional information for linking and optional verification. For primitive data constants, word 'a' will contain the actual constant value while in case of string literal it will contain a direct address to the actual literal. The word 'b' for string reference entry contains the hash table reference for use during runtime linking.

A method dispatch table is generated which contains two word entry for each method accessible in this class. Each entry is direct reference to a method code data. Depending on runtime invoke instruction, word '0' or '1' can be accessed. Its use is elaborated in Section 4.4.

3.3 Init Table

The init table generated by the Linker contains a method pointer array pointing to class initializer methods "<clinit>" - one entry for each class in the system. The array starts with a method count, and ends with "public static void main(String[] args)" entry for the start-up main method of a preselected class. The order of methods follow the hierarchy of classes in the system.

3.4 Helper Data

The helper data contains worst case execution time (WCET) constants generated for the system, which can be used for WCET analysis. It can also generate and link per-class 'helper' data to enable linking for runtime class loading.

4 Operation

This section describes object creation, field access and method invocation using the runtime data generated by the linker.

4.1 Object Instantiation

The memory management system is designed to automate creation of objects and arrays. Java instruction set includes four separate byte-codes to enable creation of objects of classes and arrays. The "new" bytecode creates objects of a given Java class while the "newarray" and "anewarray" create objects of primitive types and reference type respectively. Multi-dimensional arrays are created explicitly with the "multianewarray" bytecode.

4.2 Instantiating Classes and Arrays

The “new” bytecode, which is used to create instance of a Java class, offers a 16 bit index into the constant pool of the class of invoking method. The constant pool entry at this index provides the name of the class for which the instance has to be created.

At runtime, the memory management unit uses an instance template to create objects of any class. This template is generated by the linker for every class that it links. The instance template contains a fixed length object header followed by an integer value of length of the object. To create a new instance, the memory management system follows the logic in Figure 4. If the class is of type array indicated by a flag in class header, the header will be the header of “java.lang.Object”, followed by the number of dimensions and array type.

```
newobj*=next_obj_addr(this scope)
FOR x = 1 to header_length
newobj[x]=instance_template[x]
NEXT x
x=x+1
Length_of_object =
instance_template[x]

FOR x=1 to Length_of_object
       newobj[x]= null
NEXT x
x=x+1
//update next addr with LSBs "00"
next_obj_addr= x[31 downto 2] + 1
                                  & "00"
```

Figure 4: Create New Java Object

The object creation time can be guaranteed in exact number of clock cycles at compile time for all pre-linked Java classes. The object creation time for arrays depends on array type, number of elements and the dimensions used.

4.3 Field Access

Field access in Java objects is achieved through ‘get’ and ‘put’ bytecodes, both having static and non-static versions. For field access, the JVM passes a thisclass* pointer which represents the class data pointer for the method currently being executed. The JVM also passes a 16bit index offered by the field access opcode to the memory management system. For a non-static version of the instructions, an object pointer is also passed, which is the top of the stack of the JVM.

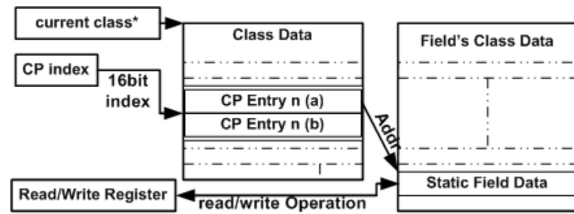

Figure 5: Static Field Access

For static field access, as shown in Figure 5 the resolved constant pool entry already has a direct pointer to the location of the static data. Therefore this operation will take two memory access cycles, a read followed by a read/write.

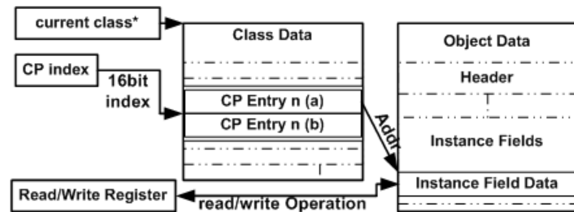

Figure 6: Instance Field Access

For non-static field access as shown in Figure 6, the resolved constant pool entry is an offset into the object data. Thus a read from class data takes place, which is followed by a read/write into the object data.

4.4 Method Invocation

The Java instruction set contains four instructions to invoke methods. For method reference entries in the constant pool pointing to a static method, the linker places a direct reference to the method code data to enable fast execution. Thus in case of the ‘invokestatic’ instruction (after accessing constant pool data), no further operation is required to start execution.

The ‘invokespecial’ and ‘invokevirtual’ methods need to be correctly resolved at runtime, and this design makes use of a Method Dispatch Table (MDT) for this purpose. During linking of each class, all methods within that class get two method addresses for each flavour (special & virtual) of the instruction in its MDT. Also, during linking all constant pool entries refering to a method (non-static) get an offset into the MDT of the class containing the actual method. The MDT is generated following Java’s inheritance model and offers a separate method code pointer for each of the two invoke instructions (special and virtual). The operation of these instructions is illustrated in Figure 7.

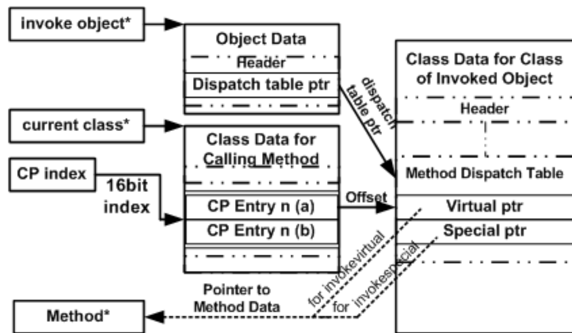

Figure 7: Resolving invoke(special and virtual)

The 'invokeinterface' instruction is considered the slowest of all the invoke instructions, as it needs to resolve a method by searching the method table of the class of object, on which the methods was invoked. This process is both time consuming and unpredictable. To address this problem, special data structures are inserted by the linker in the class data for each class.

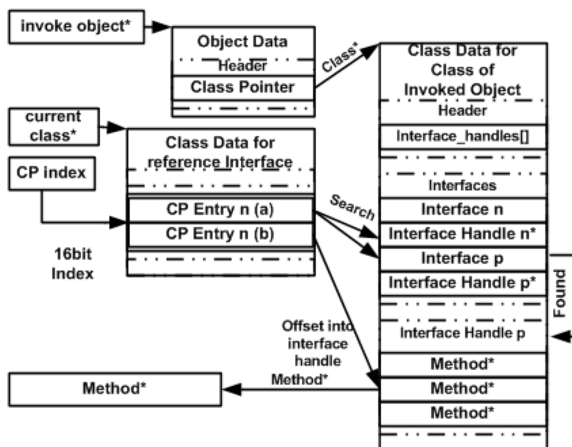

Figure 8: Resolving invokeinterface

Each class contains an interface table containing two entries for each interface implemented by the class. The first entry is the interface's class data pointer and the second entry is an offset pointing to a special structure in the class data called as an interface handle. This interface handle contains a dispatch entry for each method in the interface, which points to the correct method to be executed in the class. This data is generated by the linker for each class, which implements one or more interfaces.

Figure 8 describes the runtime operations done for 'invokeinterface' resolution. The runtime constant pool entry for an interface method reference contains an offset into an interface handle for that interface. Each class's interface handle for the same interface will be specific to that class (i.e. mapping each interface method to its implementation in reference to that class). It is notable that instead of searching the method table, it is much quicker to search the interface table and then invoke the resolved method with the help of an interface handle. The WCET for 'invokeinterface' for a whole system can be calculated with the knowledge of the maximum number of interfaces a class implements in the system. This value is not more than two or three in most cases.

5 Memory Management Scheme

The memory management system uses a segmented addressing scheme to render flexibility and simplicity in memory management. The main memory is divided into a number of segments, identified by a segment number. Any memory allocation can consist of one or more of such segments. A segment table is maintained to keep track of which segments form an allocation. To access data in any allocation, a virtual pointer scheme is used.

Figure 9: Virtual Pointer Description

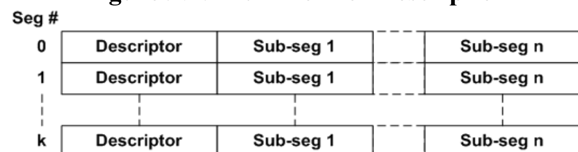

Figure 10 : Segment Table Organisation

Figure 9 illustrates a virtual pointer, which has three parts; a base segment (which is the first segment of an allocation), a segment offset (an offset into segment table, which when resolved, points to the actual segment being addressed) and an extended offset (which contains an offset into the addressed segment). Figure 10 illustrates the segment table organisation. Each segment entry has a descriptor and sub-segment table. The segment descriptor contains the descriptor of the segment (base or sub) and the count of other segments connected to the segment, to form an allocation. The sub-segments 1 to n are ordered list of segments connected to this segment to form an allocation.

An additional table is maintained, where segment number is used as an address which can read/write two words (next segment and previous segment), forming a linked list holder. The memory management system maintains a head and tail pointer for free and used memory in the system. When memory is allocated (one segment at a time), the segment at the head of the free memory linked list, is connected to the tail of the used memory linked list. At the same time the segment table entry of the base segment (of current allocation) is also updated.

The segment table is fully parallel, to ensure that all of the connected segments to a base segment can be resolved and accessed in a maximum of two cycles. An address cache unit is implemented in the memory management system to save repeated resolution of recently resolved virtual pointers. The linked list model enables freeing of any size of allocation in fixed number of clock cycles. To free a block, the head and tail of allocation is disconnected from the used memory list and connected to the tail of the free memory list.

The base segment part of the virtual pointer in an allocation also serves as the Scope ID for that allocation. In other words, each allocation will automatically become a scope. So when a reference is being checked, comparing base segment of source and destination for an assignment will immediately reveal if it is the same or a different scope. Utility signals are also generated to detect conditions like duplicate reference assignment, to ensure that correct reference count is maintained. Figure 11 shows a simplified block diagram of the complete memory management system implementation.

6 Results

The memory management system presented, guarantees predictable allocation and de-allocation times. The allocation time is proportional to the number of segments required in the allocation while the de-allocation time is a fixed number of cycles for any size of memory. Table 2 lists the number of cycles required for each memory management operation. Table 3 lists the access times for object access instructions.

Table 2: Time Grantees for Memory Management

Function	Cycles	Comments
Allocation(n)	(4+2) n	Where n is number of segments to allocate
New Scope(n)	6n + 4	Where n is number of segments to allocate
Free Scope	*5 + 1	*after finalizers have been executed
De-allocate segment	5 + 1	The effective operation is same as free scope.

All object creation, access and method invocation times are guaranteed for a linked system. Object instantiation time is solely dependent on the size of the object in words. The helper values generated by the linker contain instantiation times for each class included. Field access times for both static and non static access are of two or three cycles (an additional read/write cycle for two word values). Method invocation time for ‘invokestatic’ bytecode is just two cycles, while for ‘invokevirtual’ and ‘invokespecial’, an additional cycle is required to access the method dispatch table. Finally, the maximum latency for invokeinterface bytecode is dependent on the number of interfaces implemented by the class of the object for which the method is being called. For a class

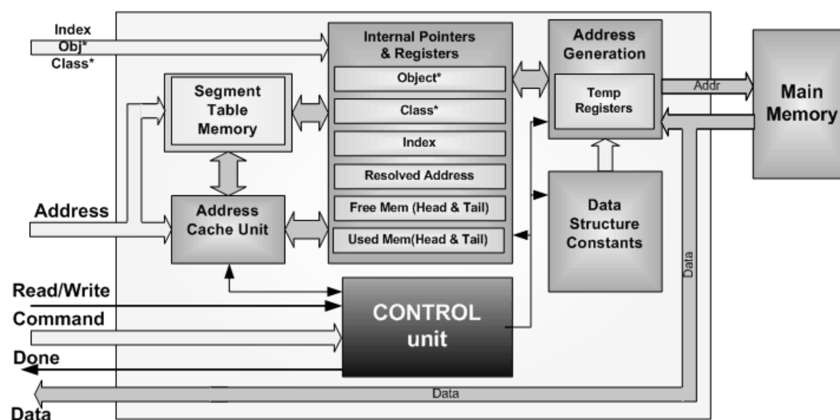

Figure 11: Memory Management System Block Diagram

which implements two interfaces; resolution of any interface reference method will take a maximum of seven cycles.

Table 3: Time Guarantees for Object Creation, Access and Method Invocation

Function	Cycles	Comments
Create Class Object (o)	4+2len(o)	Len(o) is length of object state data in words
getfield/ putfield	2/3*	For linked class(es) *When accessing double & long data
getstatic/ putstatic	2/3*	For linked class *When accessing double & long data
invokestatic	2	Cycles before method starts executing
invokevirtual	3	Includes(CP read, read dispatch-table pointer and read resolved method pointer)
invokespecial	3	Includes(CP read, read dispatch-table pointer and read resolved method pointer)
invokeinterface	Max this class = 4 + NI +1 Max any Class = 4 + S_NI Min=5	NI is number of interfaces the class implements and S_NI is maximum implemented interfaces for any class in the system

The hardware implementation of the design consumes about 19.5% of the resources, when synthesized for a Virtex 2 Pro (XC2VP7) FPGA device. This is about 6% more than our previous implementation [5], which had no object access hardware and scoped memory support. The maximum frequency estimate is about 186 MHz. These estimates are from a modified older version. A new optimised implementation is being completed, which is expected to give better area and speed results.

A linked image for classes of Java Connected, Limited Device Configuration (CLDC) version 1.1 of Java 2 Micro Edition with the linker described, fit in approximately 90Kbytes of memory, while the additional data structures used in this design (double entry MDTs, interface tables and interface handles) all together occupy just 6% of the linked memory image. Additional runtime data for a string table and a string hash table requires about 17.5KB and 7.1 KB respectively.

If a system has 4MB of total memory with a segment size of 2KB, and maximum allocation size of 32KB, it will require just over 1% of memory to form the segment table and used & free memory linked lists.

7 Conclusion

The design of a real-time memory management system for Java is presented in this paper. The design allows fast and deterministic allocation and de-allocation of memory and does not suffer from delays due to fragmented memory. It also offers creation of scoped memory areas with significant reduction of complexity involved in reference assignment checking. A dedicated linker for the system has also been presented which generates high performance data structures, for use in the memory management unit. Using these data structures and hardware based object access mechanism, all field accesses and method calls can be made with fixed or exactly predictable latencies. While this paper has described a memory management system, an instruction execution unit [6] is also being developed which will be integrated together to form a functional Java system, suitable for use in real-time applications.

References

[1] W. Foote, "Real-time Extensions to the Java Platform," presented at Fourth International Workshop on Object-Oriented Real-Time Dependable Systems, Santa Barbara, CA USA, Jan 1999.

[2] V. Narayanan, "Issues in the Design of a Java Processor Architecture," Ph.D. dissertation: University of South Florida, Tampa, 1998.

[3] G. Bollella, J. Gosling, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull, The Real-Time Specification for Java, 1st ed: Addison-Wesley, Jan 2000.

[4] M. Dham, "Byte Code Engineering Library 5.1," http://jakarta.apache.org/bcel/, 2001.

[5] A. Desai, G. Cain, J. Singh, and R. Veljanovski, "Dedicated Hardware Based Enhancements for Implementing the Java Virtual Machine," presented at The 2005 International Conference on Embedded Systems and Applications, ESA 2005, Las Vegas, Nevada, USA, 2005.

[6] A. Desai, J. Singh, and R. Veljanovski, "Modelling and Hardware Implementation of a High Performance Execution Unit for a Java Virtual Machine," presented at International Conference on Modelling and Simulation, Rouen, France, July 2005.