Deterministic Execution of Java's Primitive Bytecode Operations

Fridtjof Siebert
IPD, Universität Karlsruhe
Am Fasanengarten 5
76128 Karlsruhe, Germany
siebert@ira.uka.de

Andy Walter
Forschungszentrum Informatik (FZI)
Haid- und Neu Straße 10-14
76131 Karlsruhe, Germany
anwalt@ira.uka.de

This work was partially funded by the DFG program GRK 209

ABSTRACT

For the application of Java in realtime and safety critical domains, an analysis of the worst-case execution times of primitive Java operations is necessary. All primitive operations must either execute in constant time or have a reasonable upper bound for their execution time. The difficulties that arise for a Java virtual machine and a Java compiler in this context will be presented here. This includes the implementation of Java's class and interface model, class initialization, monitors and automatic memory management. A new Java virtual machine and compiler that solves these difficulties has been implemented and its performance has been analysed.

1. INTRODUCTION

Java [AG98] is becoming increasingly popular for software development, even in application domains well beyond the original target domain of the language. The driving force is the high productivity, code quality, security and platform-independence that typically goes along with the use of Java. Java is promoted even as a development tool for realtime critical systems that require deterministic execution [JCons00, RTJEG00], even though there are several obstacles to be overcome to achieve deterministic execution of Java code.

The most obvious source for indeterministic execution is the use of garbage collection in Java. Realtime embedded systems such as in industrial automation, avionics or automotive applications often require short response times. Blocking garbage collectors cause long pauses that are hard to predict and unacceptable for these applications.

Although incremental garbage collection techniques can help to reduce the likelihood for a blocking garbage collection pause, they can not guarantee it. It can still occur that the collector does not make sufficient progress and does not catch up with the application. Consequently, the system can fail or require long blocking pauses to recycle memory or defragment the heap.

A deterministic implementation of Java must provide means to determine worst-case execution times for Java's primitive operations. The dynamic structure of Java, with inheritance, virtual method calls and multiple-inheritance for interfaces, poses several difficulties for the implementation. The time required for calls or type checks must be limited and statically determinable.

Further difficulties are caused by the dynamic nature of Java's class loading and initialization mechanism. Finally, Java's synchronization primitives that permit to have a monitor associated with any Java object and Java's exception mechanism pose further difficulties for a deterministic implementation.

2. RELATED WORK

Research on object-oriented language implementations so far focused on high average-case runtime performance. A predictable runtime cost, that is required in realtime systems, is of little importance for classical Java applications.

For the use of Java in realtime applications, Nilsen identifies the analysis of conservative worst-case execution times for code sequences that exclude recursion and unbounded loops as a key enabling technology [Nilsen96, NR95]. This analysis is not straightforward for all of Java's primitive operations; this paper will describe how a Java implementation can permit this analysis. The other enabling technologies listed by Nilsen are execution time measurement, rate-monotonic analysis [LL73], static cyclic schedules and realtime garbage collection.

A number of publications describe technologies to efficiently implement object-oriented languages or Java specific features like monitors, improving the average-case runtime and memory costs.

Different approaches for the implementation of dynamic dispatching in the context of single- and multiple-inheritance have been proposed. Zendra et. al. suggest using specific dispatch functions instead of virtual function tables to make better use of modern processor architectures [ZCC97]. Typical implementations of *C*++ [Strou87] use several virtual function tables per object and direct references to sub-objects to implement multiple inheritance. To improve the average performance of virtual calls, inline caching has been suggested [DS84].

Yang et. al. [Yang99] and Bacon et. al. [BKMS98] presented techniques to inline monitors in objects and reduce the average runtime cost for monitor operations to a minimum. However, their approaches cannot avoid high worst-case overhead in the case of contention.

3. THE JAMAICA VIRTUAL MACHINE IMPLEMENTATION

Jamaica is a new virtual machine implementation for Java and a static Java bytecode compiler. It provides deterministic execution of the whole Java language, including deterministic garbage collection, and constant time execution of the primitive Java operations.

Jamaica uses a builder utility to create stand-alone applications out of a set of Java class files and the Jamaica virtual machine. The builder can perform smart linking and compilation of the Java application into optimized *C* code using a static Java bytecode compiler. The compiler implements several optimizations similar to those described by Weiss et. al. [Weiss98]. The generated *C* code is then translated into machine code by a C-compiler such as *gcc*.

To reduce the footprint of Java applications, interpreted compact byte code and compiled machine code can be mixed. Profiling data can be used to guide the compiler to compile only the hot spots of the applications and achieve the best speedup with a small increase in code size.

3.1 Executing Bytecodes

Most Java bytecode instructions can be implemented directly as a short sequence of machine instructions that executes in constant time when cache effects are ignored. Even in the presence of processor caches, a short worst-case execution time can be determined easily. These operations include accesses to local variables and the Java stack, arithmetic instructions, comparisons and branches.

The bytecode instructions for which a deterministic implementation is not straightforward will be described in detail in this section.

3.1.1 Loading String Constants

One needs to be careful when loading string constants with the *ldc* bytecode instruction. If this instruction is used to load a string object, this string object must have been created in advance to avoid unpredictable allocation overhead. The implementation achieves this by creating all constant string objects at class load time. During execution, loading the string's address is all that needs to be done.

3.1.2 Class Initialization

The semantics of Java require that on the first access to a static field, a static method or the first creation of an instance of a class the corresponding class be resolved and its static initializer be executed. In a standard Java implementation, the first resolution of a class also causes the class to be loaded, causing a very complex operation that might take much time. The resolution causes very long worst-case execution times of the primitive operations that might cause class resolution, while the actual execution time of these operations is typically very short once the referred class is initialized.

This problem is not easy to solve without changing Java's semantics. The Java virtual machine specification [LY99] explicitly allows early loading and resolution of classes, as long as the semantics of Java are respected. However, this does not allow early execution of static initializers, since this would change the control flow and hence the semantics of the implementation.

To avoid the overhead of loading and linking a class during class resolution, the Jamaica virtual machine recursively loads and links all classes that are referenced by the root class at system initialization. If later during execution of the system additional classes are loaded using methods like *java.lang.Class.forName()* or the reflection API, this process is repeated and all referenced classes are loaded as well.

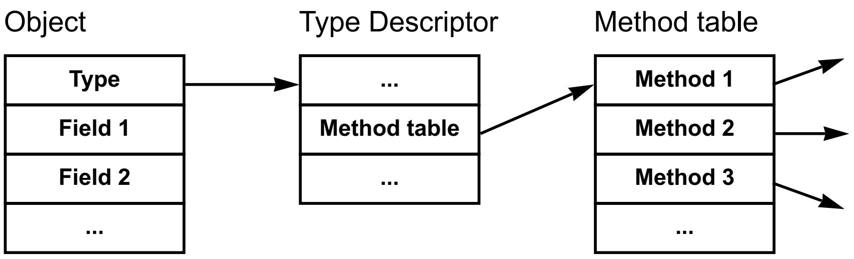

Figure 1: Method table used for dynamic binding of virtual calls

What is left at the first reference to a class is the execution of its static initializer. The user is free to perform arbitrarily complex calculations within the static initializer, so a worst-case execution time for this operation can not be guaranteed by the Java implementation unless the static initializer is relatively simple, e.g., code that contains no method calls or loops.

Even for simple static initializers, the call overhead causes fairly bad worst-case execution times for simple operations like an access to a static field. The user can avoid this overhead by explicitly causing early initialization of the referred class during system startup.

The resulting overhead for operations that cause class resolution is then reduced to the overhead of checking the initialization state of the class. This can be done by reading a single field in the class' descriptor and a conditional branch.

We plan to improve the Jamaica compiler such that explicit early initialization can be detected statically such that the test can be avoided in most cases.

3.1.3 Method Invocation

The invocation of methods in the context of class extension and interface implementation is another difficulty for a deterministic implementation of Java. There are four different bytecode instructions for the invocation of methods.

Static Calls

Two call instruction avoid dynamic binding altogether: *invoke_static* and *invoke_special*. The first one is used to call static methods, while *invoke_special* is used for instance methods, but uses static binding instead of dynamic binding (this is used for object initialization or when explicit calls to a certain class' method is needed as in *super.method()*). The lack of dynamic binding semantics in these two bytecode instructions makes the implementation straightforward. Control flow can directly switch to the called method since the target method is known.

Virtual Calls

A call using *invoke_virtual* is slightly more difficult to implement since the dynamic type of the target object needs to be taken into account. If the called method was redefined by the dynamic type of the target, that redefined method needs to be called.

The standard method of implementation provides deterministic execution time. A virtual call is implemented with a method table that is part of the type descriptor of each object (**Figure 1**). Each method is assigned a unique index in the method table. Inherited methods keep their unique indices in the new class. Methods defined within a class are assigned the next available index, while methods that redefine inherited methods inherit the index of the original method. Every method is recorded in the method table at its index, while inherited methods that were redefined are replaced by their redefined versions.

This technique permits one to find the target method of a virtual call very efficiently. All that is required is a lookup in the method table at the method's index. If the method has been redefined, the redefined method's entry will use the same index as the original method and the redefined method will be found. Using this technique, a virtual call can be performed in constant time.

Interface Calls

The most difficult to implement calls are calls to interface methods via the bytecode operation *invoke_interface*. The reason is that multiple inheritance

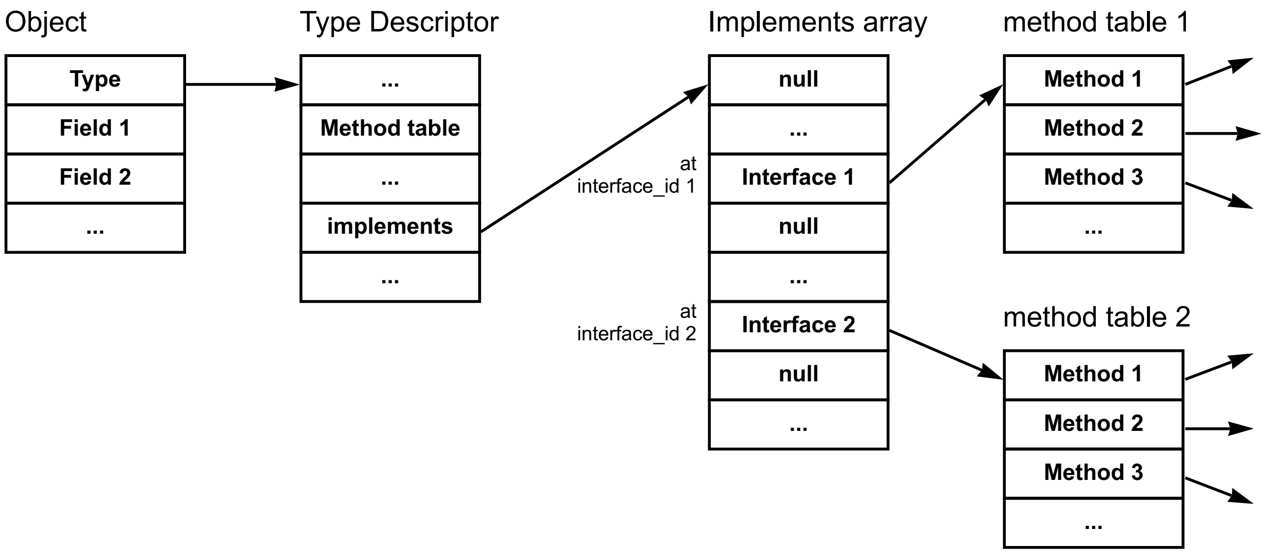

Figure 2: Interface call using implements array

makes it impossible to assign constant indices to the methods as in the case of virtual calls. A dynamic search for the called method is typically made in this case. This causes a call overhead that depends on the number of methods in a class or the number of interfaces implemented by a class. Therefore calls to interface methods are significantly less efficient in many Java implementations and the execution time depends on the structure of the target class.

The representation of classes and interfaces that was chosen for the Jamaica implementation permits constant-time calls to interface methods. Every interface is assigned a unique identifier *interface_id*, starting at *0* for the first interface loaded. Each method within an interface is assigned an *interface_method_index* similar to the methods of normal classes. The class descriptor of every class that implements one or several interfaces contains a reference to an array of references. This array is referred to as *implements* array. For each interface that is implemented, the *implements* array contains a valid reference at the interface' *interface_id*. This reference points to an array of the methods defined in the interface and implemented in the class, with each method at its *interface_method_index* (**Figure 2**). All entries in the *implements* array that correspond to interfaces that are not implemented by the class are set to *null*. For the call of an interface method, all that is needed is to read the *implements* array from the target object's type descriptor, index this array at the interface' *interface_id* and finally read the method at the *interface_method_index*. Compared to virtual methods, there is only one additional indirection needed, calls to interface methods are performed in constant time.

Nevertheless, there is an important disadvantage of this approach: the size of the *implements* array is linear in the number *i* of interfaces, and the total memory required for a system with *c* classes is therefore in *O(c·i)*. The memory usage is quadratic in the size of the code. With the applications executed using Jamaica, this has not caused any difficulties, but it might be a problem for very large applications.

The memory overhead can be reduced using some simple optimizations. It is sufficient if the size of the *implements* array is reduced to *interface_id*+1 for the largest *interface_id* of all the interfaces implemented. In addition, all classes that do not implement any interfaces can share one single empty *implements* array. Jamaica makes use of these optimizations.

Dynamic loading of interface classes can also be handled using this mechanism. New interface classes will be assigned new *interface_ids* and classes implementing this interfaces will get sufficiently large *implements* arrays. In the context of class garbage collection, care must be taken to ensure that unused *interface_ids* will be reused, so that the size of the *implements* arrays does not grow without limit.

One can imagine that the typically high additional overhead for calls due to multiple inheritance might be the only reason for the existence of interfaces.

Using the scheme presented here, the overhead is minimal compared to a virtual call using single inheritance. If Java allowed multiple inheritance for normal classes, there would be no need for interfaces at all. A single concept, the class, could be used to model interfaces and classes. This would remove the often difficult decision whether to model a concept using an abstract class or an interface. The former is more powerful by allowing the implementation of some methods, while the latter is more flexible and allows multiple inheritance. Object-oriented programming languages like Eiffel have shown that a single concept using multiple inheritance is sufficient and that efficient implementation of multiple inheritance is possible [Meyer92].

Additional Call and Return Overhead

Apart from the overhead needed to determine which method is called, a call requires setting up a new stack frame. Later, this stack frame has to be removed when the method returns. Unlike in languages like *C* or *C++*, the creation and removal of the stack frame is not a constant time operation. The reason is that reference values on the stack frame must be known to the garbage collector. Each entry in the stack frame that is reserved for a memory reference must be at least initialized at the start of a method call and also perhaps cleaned up at the end of the call. Consequently, the overhead for creation or removal of a stack frame is linear in its size.

3.1.4 Type Checking

Another problem area for a deterministic implementation that is closely related to dynamic calls is dynamic type checking. There are two situations in Java code that require dynamic determination of the type of a reference: explicit type tests using the *instanceof* operator and type casts. Consequently, there are two bytecode operations, *instanceof* and *checkcast*, for these two purposes. The first of these, *instanceof,* tests if a referenced object is of a certain type and produces a *boolean* result, while *checkcast* causes an exception if the type test fails, and does nothing otherwise. Furthermore, there are three categories of types in Java that have to be treated differently in the type check: classes, interfaces, and arrays.

Type Checking for Classes

The most common case for a type check is checking whether or not a referenced object is an instance of a certain class. As an example, assume the following *if*-statement.

```
if (r instanceof A) {
  System.out.
    print("r is of class A");
}
```

The straightforward implementation of this type check would traverse the inheritance-chain of the object referenced by *r* until either class *A* is found or the root object *java.lang.Object* has been reached. This implementation requires time linear in the depth of the inheritance tree, a worst-case execution time for the type check is difficult to determine.

A simple modification of the representation of inherited classes permits constant-time type checking for classes. Every class has a fixed position in the inheritance tree and a fixed distance to class *java.lang.Object*. All classes in the inheritance tree that are on the path from a class *C* to *java.lang.Object* are referred to as *C*'s ancestors. *C's* ancestors include the classes *C* and *java.lang.Object*. The number of ancestors of class *C* is the *depth* of *C* in the inheritance tree. Any class descriptor can now be equipped with a reference to an array of all ancestors, as shown in **Figure 3**. Each ancestor that resides at position *depth* in the inheritance tree is stored at position *depth-1* in this *ancestors* array. Now, the type check can be done in constant time. All that is needed is a check of the *ancestors* array's entry at the position corresponding to the class' depth.

C-code that performs the *instanceof*-check shown above would look like this.

```
if ((r!=null              )
 && (r->type->
        ancestors.length>=A->depth)
 && (r->type->
        ancestors[A->depth-1] == A))
{
  System.out.
    print("r is of class A");
}
```

The code requires reading the referenced object's class descriptor, the *ancestors* array, the array length and one element in the array. The *depth* of class *A* and the class descriptor of class *A* are also required in this test, but these values are runtime constants and can be inlined by a compiler.

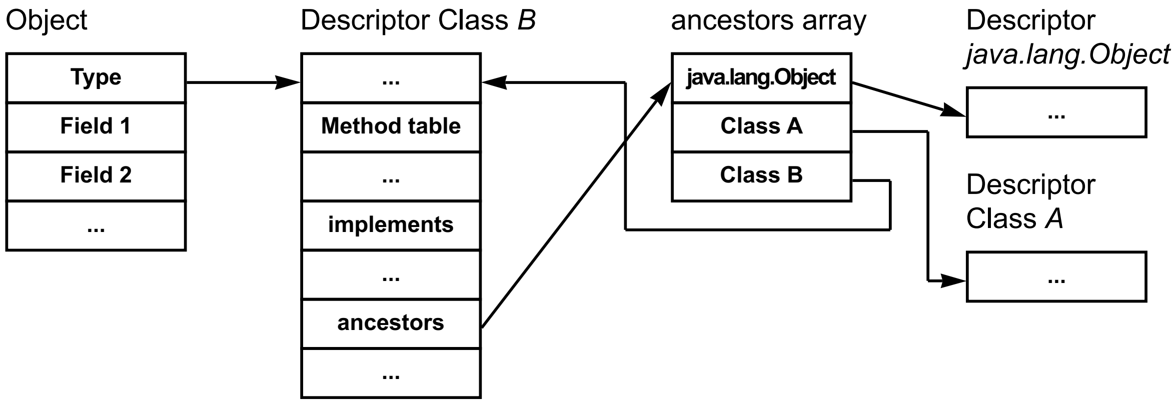

Figure 3: Type checking for classes using ancestors array. Class *B* extends class *A* which inherits directly from class *java.lang.Object.*

Type Checking for Interfaces

The *ancestors* array approach used for class types cannot be used directly for interfaces since a class might implement several interfaces that reside at the same *depth* in the inheritance tree. What is needed is an array with a unique entry for every interface that is implemented by a class. Fortunately, such an array was already introduced for the efficient implementation of interface calls: the *implements* array.

For a type check of the form

```
if (r instanceof I) {
  System.out.
    print(“r is of interface I“);
}
```

all that is needed is to check whether the *implements* array of *r* has an entry for interface *I*. In C-like code, this might look as follows.

```
if ((r!=null                    )
 && (r->type->
        implements.length >  I->id)
 && (r->type->
        implements[I->id] != null ))
{
  System.out.
    print(“r is of interface I“);
}
```

The code requires reading the class descriptor reference, the implements array, its length and the entry corresponding to the interface *I*. The *interface_id* that is required is a runtime constant.

Type Checking for Arrays

The type checking semantics for arrays are defined recursively. The examined type must be an array. If this is the case, the type checking continues for the element type of the array. A straightforward implementation of this definition requires execution time linear in the number of dimensions of the array. Nevertheless, a constant-time implementation of type checking for multidimensional arrays is possible. One need only store the dimension count and a reference to the final non-array element type with the array type.

This can be illustrated using the following code sequence.

```
if (r instanceof A[][][]) {
  System.out.
    print(“r is of type A[][][]“);
}
```

All that is required for this test is to check whether the object referred to by *r* is an array of dimension three and the base element type *A*. The C-code might look like this:

```
if ((r!=null                      )
 && (r->type->dimensions == 3      )
 && (r->type->elemnt->
        ancestors.length >=A->depth)
 && (r->type->elemnt->
        ancestors[A->depth-1] == A))
{
  System.out.
    print(“r is of type A[][][]“);
}
```

The code checks the correct number of dimensions of the array and then does a type check for the element class as described above.

Care needs to be taken when checking arrays of class *java.lang.Object* or interfaces *java.lang.Clo-*

neable and *java.io.Serializable.* The class *java.lang. Object* is an ancestor of all arrays and the two interfaces are implemented by all arrays. To implement the type check for these arrays correctly, the check must additionally succeed whenever the dimension of the examined type is higher than that with which the array type it is compared.

3.1.5 Switch Statement

The Java bytecode provides two different operations for *switch*-statements: *tableswitch* and *lookupswitch*. For a *switch*-statement with *case*-entries that are similar values, *tableswitch* is used. It permits a constant-time table lookup for the statement sequence that needs to be executed.

For *case*-entries that extend over a large range of values with unused intervals *tableswitch* would require a large table. In this case, *lookupswitch* is used in the bytecode. This instruction uses a sorted table of *case*-value and target address pairs. The lookup requires a binary search of this table, hence the runtime is logarithmic in the number of *case*-labels in the switch statement.

The Java implementation of the interpreter has little choice here but to implement the binary search, converting a *lookupswitch* into a constant-time *tableswitch* might require too much memory for the table. The user of the system must therefore be careful when using *switch*-statements with *case*-entries that stretch over a large range of values.

A Java compiler that generates native code can use perfect hashing here. During compilation time, a conflict-free hash function is computed which maps the case-values to target addresses. The jump is then executed in $O(1)$. As shown by Mehlhorn, such a hash table of size $3n$, where n is the number of case-labels, can be computed in cubic time [Mehlh84]. Future versions of Jamaica that generate native code will use perfect hashing here. The currently generated *C* code uses *C*'s switch statement and relies on the *C* compiler's implementation.

3.1.6 Memory Allocation

The most difficult problem to be solved by a deterministic implementation of Java is to provide deterministic behaviour of dynamic memory allocation. The implementation has to guarantee a hard upper bound for the execution time of an allocation. But this is not sufficient, it also has to guarantee that the garbage collector recycles memory sufficiently quickly for the application not to run out of memory. Finally, the implementation must guarantee that fragmentation or conservative scanning techniques do not cause the loss of memory in a way that allocations cannot be satisfied. The implementation consequently uses an exact garbage collector that has accurate information on all reference values in objects, local variables, stacks and processor registers.

To avoid fragmentation, compacting or moving garbage collection techniques are typically employed. These techniques nevertheless cause several difficulties for an efficient implementation. For example, references to moved objects must be updated to the new location of a moved object. This updating must include references in local variables or processor registers.

The Jamaica implementation uses a new approach to avoid fragmentation altogether. The heap is regarded as an array of blocks of a fixed size (typically 32 bytes per block). On allocation of an object, at least one such block is used. If this is not sufficient, a linear list of possibly non-contiguous blocks is used to represent the object. Arrays are represented as a tree of blocks with the array elements just in the leaf nodes. The use of this object and array model allows the use of non-contiguous regions of memory for allocation. There is no need to defragment memory and move objects. This technique to avoid fragmentation has been described in detail in an earlier publication [Siebert00].

In classic Java implementations, the garbage collection activities occur unpredictably whenever the system memory runs low or certain thresholds are reached. Additionally, these implementations typically cannot guarantee that the garbage collector performs sufficient recycling work to catch up with the allocation of the application. This approach is not applicable for a deterministic implementation.

Jamaica couples garbage collection activity with allocation. Whenever a block of memory is allocated, a certain number of blocks are scanned by the garbage collector. As has been shown earlier [Siebert98], this approach can guarantee sufficient garbage collection progress for the system not to run out of memory and it can guarantee upper bounds for the amount of garbage collection work that is required

k:	0.0	0.3	0.4	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95	0.975	1.0
P_{max}**:**	1.0	6.173	8.085	10.67	14.47	20.71	25.67	33.10	45.45	70.11	144.0	291.8	∞

Table 1: Upper bound for required GC progress per unit of memory allocated

for the allocation of one block of memory as long as the amount of reachable memory used by the application is limited.

For an application that uses not more than a fraction *k* of the total heap size as reachable memory, the upper bound P_{max} of the number of units of garbage collection work that need to be performed for the allocation of one block of memory can be determined statically. Several values for *k* and the corresponding values of P_{max} are presented in **Table 1**. On the PowerPC architecture, one unit of garbage collection work for a block size of 32 bytes corresponds to 266 machine instructions in the worst case. This permits the determination of a worst-case execution time for the allocation time of Java objects and arrays.

3.1.7 Memory Accesses

The use of fixed size blocks to represent Java objects and arrays has an important impact on the code that is required to access object fields and array elements.

Field Accesses

Since a linear list of blocks is used to represent objects, the number of memory references required is linear in the offset of the field within the object. Fields that reside in the first block can be accessed using a single memory access, while fields in the second block require two memory accesses, etc.

Fortunately, most Java objects are very small and even for larger objects the fields that are accessed most frequently are typically the first fields with the lowest offsets [Siebert00]. For any field, the position is known statically and hence the number of memory accesses required can be determined statically.

Array Accesses

With array accesses, the situation is a bit more complex since arrays are represented as trees. The trees use the highest branching factor allowed by the block size. For a block size of 32 bytes on a 32-bit system, this means that the branching factor is 8. Even very large arrays can be represented in a fairly shallow tree. For a given system with a limited heap size *h*, the maximum depth d_{max} for these trees can be determined statically using the following term (for a block size of 32 bytes).

$$d_{max} = \lceil ln_8(h / 32) \rceil$$

For a heap of 32MB the resulting maximal depth is 7. For an array access, one additional memory access is required to read the array's depth and one reference is needed to read the actual element. So the total number of memory references is d_{max}+2, or 9 for a heap size of 32MB.

This enables the determination of a worst-case execution time for array accesses. Even though this time is much higher than that of a classical linear array representation, it will be shown below that the overall performance of the system is comparable to that of traditional Java implementations.

3.2 Monitors

Several publications have presented efficient implementations of Java monitors that reduce the memory overhead of inlined monitors and the runtime overhead in the most common cases. Yang et. al. [Yang99] propose inlining the monitor and reserving one word per object for the monitor. This word consists of three sub-word sized integers that represent the nest count, the owner thread and a list of waiting threads. The nodes of this list are stored in a hashtable and they are only used if threads are actually waiting for a monitor.

Bacon et. al. [BKMS98] reserve 24 bits per object for the monitor. They distinguish two different representations for monitors: inlined and inflated. Inlined monitors use the monitor value to store an identifier for the owner thread and a nest count. Inflated monitors use the 24 bits as a monitor id that functions as a reference to a monitor object on the heap. Monitors that are never subject to contention remain in inlined representation, while monitors that are subject to contention will be converted to their inflated representation and will remain in this representation until the object dies.

The disadvantage of these monitor representations is their unpredictable runtime behaviour due to the use of heap allocation or a hashtable for monitors that are subject to contention.

The Jamaica monitor implementation avoids the need to dynamically allocate monitor storage on the heap or stack altogether. Monitors are always inlined, using 16 or 32 bits per object (16 bits are sufficient in systems with few threads and a limited nest count). As long as no threads are waiting for a monitor, it is sufficient to record the owning thread's identifier and the nest count in the inlined monitor (**Figure 4**). As soon as a thread tries to acquire a monitor that is owned by a different thread, a queue of waiting threads needs to be created.

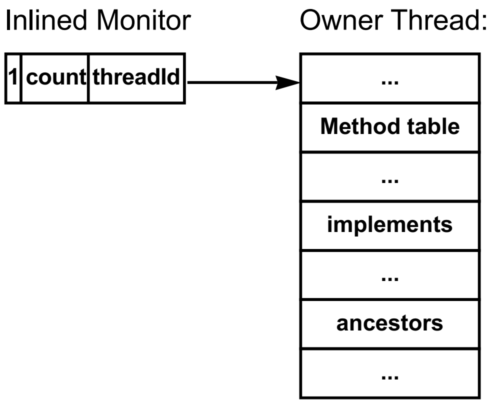

Figure 4: Inlined monitor owned by one thread.

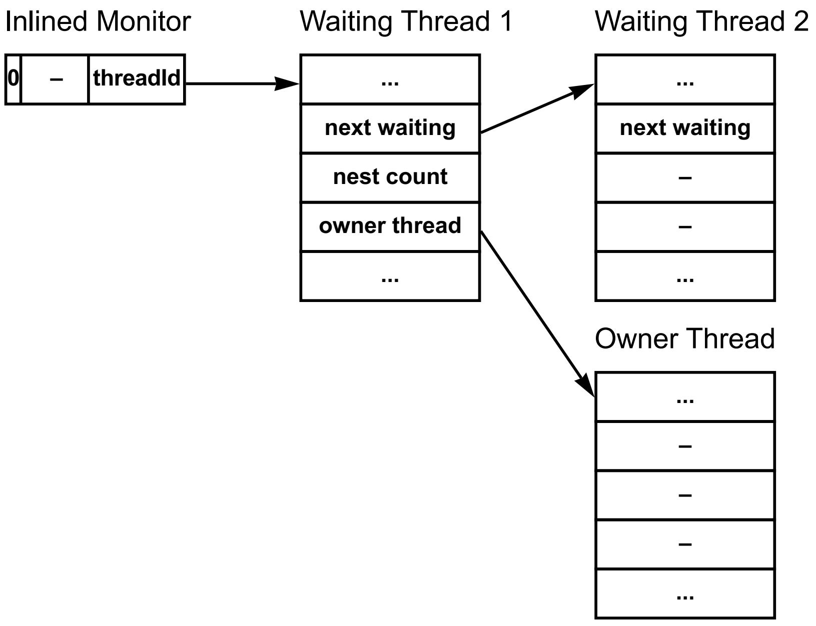

Figure 5: Monitor owned by one thread with two waiting threads.

An important observation one can make is that no thread can ever be waiting for more than a single monitor. Hence, instead of creating independent node objects for the waiting queue, one might as well reserve some additional fields in the thread object itself and create a queue of thread objects for all waiting threads. Since a thread might own several monitors simultaneously, the owning thread itself must not be part of this queue, only the waiting threads can be linked to the monitor. One bit in the inlined monitor indicates that a waiting queue exists. In this case, the nest count and owning thread's identifier are copied into reserved fields of the threads in the waiting queue. The identifier of the first thread in the queue is stored in the inlined monitor (**Figure 5**).

The result is a monitor implementation that is very efficient for the most frequent cases of monitor operations. Entering a monitor that is not owned by any thread or that has already been acquired by the current thread and exiting a monitor that has no other threads waiting. The operations that involve waiting threads do not require dynamic creation of a monitor object, all that needs to be done is queuing and unqueuing the waiting threads and performing the actual wait or resume.

3.3 Exceptions

Java's exception mechanism enables control flow from an active method back to a method with an exception handler that lies an arbitrary number of stack frames above the current position in the call chain. Throwing an exception that is handled by another method therefore requires removal of all active stack frames that lie in between the method causing the exception and the method handling it. This includes removing of all local references in these stack frames from the garbage collector's root set, which is an

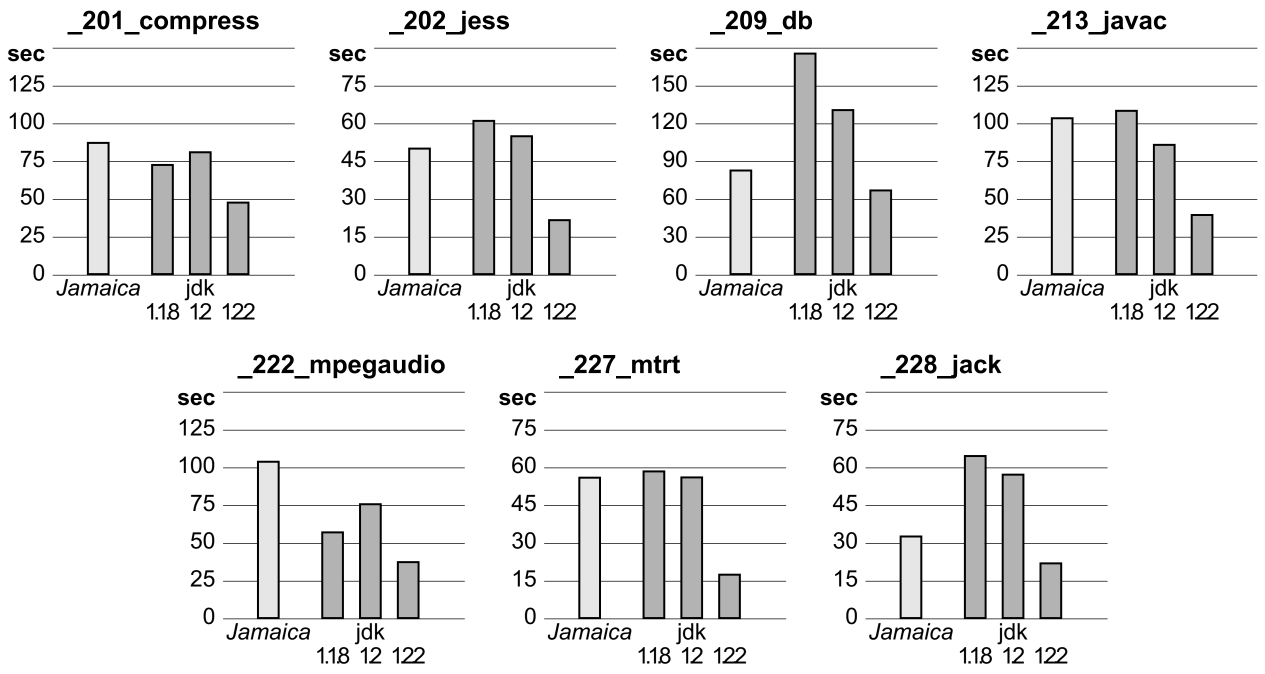

Figure 6: Runtime performance of the SPECjvm98 benchmarks using Jamaica and JDK 1.1.8, 1.2 and 1.2.2

operation linear in the number of active stack frames and in the number of root references.

The execution time for an exception is therefore not constant and it is at least difficult to find a constant-time implementation for exception handling. Using a mechanism like *C*'s *setjmp* and *longjmp* enables constant-time change of the control flow that is required for exceptions, but it does not provide means to remove the information required for accurate garbage collection.

In the Jamaica implementation, the execution time of throwing an exception is therefore linear in the size of the stack frames between the exception throwing point and the corresponding exception handler. Since exceptions are not supposed to be used for normal control flow, but for exceptional control flow only, this behaviour should be acceptable even when deterministic execution in the non-exceptional case is required.

4. PERFORMANCE COMPARISON

To evaluate the overall performance of the deterministic Java implementation Jamaica and to compare it to traditional Java implementations, the SPECjvm98 benchmark suite [SPEC98] has been run using Jamaica and several versions of SUN's JDK [SUN99, SUN00] with JIT compiler enabled (*JDK 1.1.8*; *Classic VM build JDK-1.2-V, green threads, sunwjit*; and *Solaris VM build Solaris_JDK_ 1.2.2_05, native threads, sunwjit*). Only one test from the benchmark suite, *_200_check*, was excluded from the analysis since it is not intended for performance measurements but to check the correctness of the implementation (and Jamaica passes this test).

For execution, the test programs were compiled and smart linked using the Jamaica builder. The programs were then executed on a single processor (333 MHz UltraSPARC-IIi) SUN Ultra 5/10 machine equipped with 256MB of RAM running SunOS 5.7.

The results of the performance measurements are shown in **Figure 6**. For the measurements, the heap size was set to 32MB for all tests but *_201_compress*. This test required more memory to execute and was run with a heap size of 64MB.

Compared to Sun's implementation, the performance of the deterministic implementation is similar to that of JDK 1.1.8 or 1.2, while the performance of JDK 1.2.2 was improved significantly. Since Jamaica is still a very young implementation and not much effort has been spent on improving and tuning the compiler's optimization techniques, one can expect that better optimization in the compiler will permit to improve the performance of Jamaica further. Another source for future performance enhancements will be direct generation of machine code instead of using *C* as an intermediate language as is done currently. Using *C* does not permit optimal code selection for primitives like write-barrier code or garbage collector root reference information. Direct generation of machine code permits selection of better code for these primitives, e.g., by assigning certain registers for specific purposes.

5. CONCLUSIONS

In this paper the Jamaica implementation of Java has been presented. The deterministic implementation of Java's primitive operations in this implementation has been explained. The implementation permits static analysis of the execution time of its primitive operations that is not possible with current Java implementations.

The performance of the implementation has then been measured and compared to different versions of Sun's JDK implementation using the SPECjvm98 benchmark suite. The results show that performance that is comparable to Sun's implementations can be reached. These results encourage us to further improve our implementation and to further reduce the performance gap compared to non-deterministic Java implementations.

6. FUTURE WORK

The deterministic implementation of a programming language's primitive operations provides only the basis for further analysis of the code. Tools for automatic determination of worst-case execution times can be build on top of this. For an accurate analysis that is not too conservative, mechanisms to model the system's cache memories and the effects of modern superscalar processors on the execution time need to be developed.

7. AVAILABILITY

The described Java implementation is available for academic and non-academic purposes. Please contact the authors or visit the web-site at *http://www.aicas.com* to obtain further information.

REFERENCES

[AG98] Ken Arnold and James Gosling: *The Java Programming Language*, 2nd edition, Addison Wesley, 1998

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: *Thin Locks: Featherweight Synchronization for Java*, PLDI, 1998

[DS84] L. Peter Deutsch and Allan M. Schiffman: *Efficient Implementation of the Smalltalk-80 System*, Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, January, 1984

[JCons99] J-Consortium: *Real Time Core Extension for the Java Platform*, Draft International J-Consortium Specification, V1.0.14, September 2, 2000

[LL73] C. L. Liu and James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, January 1973, pp. 46-61

[LY99] Tim Lindholm and Frank Yellin: *The Java Virtual Machine Specification*, Second Edition, Sun Microsystems, 1999

[Mehlh84] Kurt Mehlhorn: *Data Stuctures and Algorithms 1: Sorting and Searching*, Springer Verlag, Berlin, 1984

[Meyer92] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall International (UK), Hertforshire, 1992

[Nilsen96] Kelvin Nilsen: *Java for Real-Time*, Real-Time Systems, 11, pp. 197-205, Boston, 1996

[NR95] K.D. Nilsen and B. Rygg: *Worst-Case Execution Time Analysis on Modern processors*, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, San Diego, 1995

[RTJEG99] The Real-Time Java Experts Group: *Real Time Specification for Java*, Addison-Wesley, 2000, http://www.rtj.org

[Siebert98] Fridtjof Siebert: *Guaranteeing non-disruptiveness and real-time Deadlines in* *an Incremental Garbage Collector (corrected version)*, International Symposium on Memory Management (ISMM'98), Vancouver, 1998, corrected version available at *http://www.fridi.de*

[Siebert00] Fridtjof Siebert: *Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java*, Compilers, Architectures and Synthesis of Embedded Systems (CASES'00), San Jose, November 2000

[Strou87] Bjarne Stroustrup: *Multiple Inheritance for C++*, Proceedings of the European Unix Users Group Conference, pp. 189-207, Helsinki, May, 1987

[SPEC98] *SPECjvm98 benchmarks suite, V1.03*, Standard Performance Evaluation Corporation, July 30, 1998

[SUN99] *Java Development Kit 1.1.8*, SUN Microsystems Inc., 1999

[SUN00] *Java Development Kit 1.2.2*, SUN Microsystems Inc., 2000

[Weiss98] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vanial Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler: *TurboJ, a Bytecode-to-Native Compiler*, Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Montreal, in Lecture Notes in Computer Science 1474, Springer, June 1998

[Yang99] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioglu, Erik Altmann: *Leightweight Monitor for Java VM*, ACM Computer Architecture News, Vol 27/1, March 1999.

[ZCC97] Olivier Zendra, Dominique Colnet and Suzanne Collin: *Efficient Dynamic Dispatch without Virtual Function Tables*, OOPSLA, 1997