A Java Virtual Machine Architecture for Very Small Devices

Nik Shaylor
Sun Microsystems Research Laboratories
2600 Casey Avenue
Mountain View, CA 94043
USA
nik.shaylor@sun.com

Douglas N. Simon
Sun Microsystems Research Laboratories
2600 Casey Avenue
Mountain View, CA 94043
USA
doug.simon@sun.com

William R. Bush
Sun Microsystems Research Laboratories
2600 Casey Avenue
Mountain View, CA 94043
USA
bill.bush@sun.com

ABSTRACT
The smallest complete Java™ virtual machine implementations in use today are based on the CLDC standard and are deployed in mobile phones and PDAs. These implementations require several tens of kilobytes. Smaller Java-like implementations also exist, but these involve compromises in Java semantics. This paper describes a JVM™ architecture designed for very small devices. It supports all the CLDC Java platform semantics, including exact garbage collection, dynamic class loading, and verification. For portability and ease of debugging, the entire system is written in the Java language, with key components automatically translated into C and compiled for the target device. The resulting system will run on the next generation of smart cards, and has performance comparable to the reference CLDC implementation available from Sun™.

Categories and Subject Descriptors
D.3.4 **Programming Languages**: Processors – *Interpreters, Optimization, Preprocessors, Run-time environments.*

General Terms: Languages.

Keywords: Java, CLDC, JVM, next generation smart cards, limited-memory devices.

1. INTRODUCTION
The work described here is a continuation in spirit of the Spotless project [7], begun in 1998. The goal then was to build a small but complete Java virtual machine, with the Palm Pilot the target platform. The resulting artifact turned into the KVM, which became the basis of the CLDC standard [10].

The main goal of the present effort was once again to build a small Java virtual machine, but one that is smaller and more mature (CLDC compliant, with verification and exact garbage collection). The target platform is the next generation of smart cards, which have 32-bit processors, but may have no more than 8 KB of RAM, 32 KB of non-volatile memory (NVM, typically EEPROM), and 160 KB of ROM. Such a VM would obviously also be suitable for other embedded devices with similar characteristics.

A secondary goal was to write the system as much as possible in the Java language, for both portability and ease of debugging. The system is therefore called Squawk, in homage to the Squeak Smalltalk system [12].

LCTES'03, June 11-13, 2003, San Diego, California, USA

2. BACKGROUND
The size of Java classfiles has long been recognized as an issue, especially for embedded devices. A number of efforts have been made to reduce the size, both as a transmission format and as an execution format. Pugh [4] developed techniques for compressing classfile components for transmission, and achieved sizes ranging from 17% to 49% of the comparable JAR files. Rayside et al. [5], in contrast, focused on execution format, specifically reducing constant pool size and code size. Reductions in JAR file size of roughly 50% were obtained through optimizations of the constant pool, and smaller improvements were realized through code optimizations. Clausen et al. [2] developed a compressed bytecode scheme using macros, achieving space savings of around 30%, but at a runtime cost of up to 30%. Tip et al. [8] developed techniques for extracting only the components necessary for a particular application from a set of classfiles, with the resulting archives reduced to 37.5% of their original size.

A primary goal of the above techniques was to perturb the Java execution engine as little as possible. In contrast, the Java Card paradigm introduces a muscular transformer between classfiles and interpreter, which enables a different, space-optimized interpreter.

Efforts have been made to implement Java in the Java language: The JavaInJava system [6] was an early effort that encountered performance issues and ran roughly 3 orders of magnitude slower than a native implementation. The Jalapeno/Jikes [1][15] system has extensively explored the optimization technology needed to produce a high-performance system. The Joeq system [16], an

open source effort, is also focusing on performance. Both Jikes and Joeq are relatively large.

There are a number of commercial Java implementations targeted for small devices. Information on the size characteristics of some of these implementations is available: Waba [17] requires less than 64 KB of RAM to run (although it is not Java-compliant); Esmertec's JBED ME [13] has a footprint of 210 KB; and Aplix's JBlend [14] can run in less than 30 KB of RAM, but 256 KB is recommended. IBM's WebSphere Micro Environment [18] has interesting characteristics, but its size characteristics are unknown.

3. DESIGN

The project's main goal of building a CLDC-compliant system on a small device led to a few straightforward consequent goals, namely minimizing the size of transmitted classfiles, the RAM needed for classfile loading, the size of loaded classfiles, the RAM required during execution, and the size of the interpreter and memory system. In addition, the system had to effectively deal with NVM.

These goals were in turn realized through a few clear design choices.

- Standard CLDC classfiles are too large and complex to process on a device with 8K of RAM. Employing a Java Card™ technique [11], they are instead preprocessed off-device and packaged into a smaller representation, called a *suite*, which then can be verified and loaded on-device in 8 KB.
- Certain types of code make verification and garbage collection substantially more complicated. The classfile preprocessor, called the translator, identifies these problems and produces simple, semantically equivalent suite code.
- The standard bytecode set is not as space efficient as it could be. The translator transforms the bytecodes into a more compact form.
- The standard classfile contains symbolic information for resolving references to various entities. This feature makes classfiles larger, slows down initial execution (when these symbolic references are resolved), and precludes the efficient execution of classfiles in NVM. The Squawk system resolves the references before execution, either during off-device translation or during the installation of suites on the device (which is allowed by the CLDC standard) into NVM.
- The standard mechanism for constructing and initializing objects complicates verification, which as a result requires too much space on a small device. The Squawk translator recasts object creation into an equivalent form that can be verified more efficiently.
- As with other small devices, the next generation Java Card has three memory spaces with distinct characteristics. The Squawk system uses a region-aware memory manager and uses NVM whenever reasonable for persistent data (such as classes and methods).

The resulting design is not only small enough to run on a small device, but is relatively fast and portable because of its simplicity. The rest of this section presents details of the above features.

More information can be found in the Squawk specification [9].

3.1 Code Structure

Bytecode verification and exact garbage collection are made substantially more complex, and thus more difficult on a small device, due to the code allowed on a standard JVM. The general problem is that pointers are hard to find and track. Specifically:

- Local variables are allowed to change from holding values of one type to values of another at arbitrary places in a method.
- The evaluation stack may contain an arbitrary number of entries at a basic block boundary.
- Methods can be called when there are arbitrary entries on the evaluation stack.

The CLDC specification addressed these issues by introducing a classfile attribute, called a *stackmap*, that identifies the types of local variables and stack entries in a method. However, this mechanism is still too complex and memory intensive for the devices intended for Squawk.

These problems are solved in Squawk by placing concomitant restrictions on the code executed by the interpreter. These are:

- A local variable can only hold one type. This restriction simplifies both verification and garbage collection. Pointers held in local variables can always be found unambiguously.
- The evaluation stack must be empty at the end of a basic block. This restriction simplifies verification because no stack merging analysis needs to be done.
- Bytecodes that can trigger a garbage collection may only be executed when the evaluation stack contains only the operands for those bytecodes. This restriction (in conjunction with the first two) saves space by requiring only a single bit vector to locate all the pointers in a method's activation record (essentially requiring only a stackmap per method instead of a stackmap per basic block).

The Squawk system implements these restrictions in the translator that converts classfiles to suites. The translator (a) creates extra local variables for existing local variables that are reused and (b) creates extra local variables to hold stack values that need to be stored at basic block boundaries and at instructions that can trigger a garbage collection. It also inserts additional load and store instructions to manage the values put in these extra variables.

The first transformation proved to be more complex than anticipated because it relied on the liveness information implicit in the CLDC stackmaps to identify reuse of local variables. However, the stackmaps produced by the current version of the CLDC are not completely accurate; they are occasionally describe a local variable as live at a basic block boundary even though the variable is never used from that point on. The translator therefore performs its own liveness analysis.

These transformations can in theory substantially increase the number of variables required for a method. However, the translator also includes an aggressive register allocation algorithm to reduce the final number of local variables required by a method. In practice, the increase in the number of local variables is relatively small. For the current Java Card 2.0 core API there was a 6% increase in local variables. Due to the compact bytecode instruction set used in suites, the average size of

methods decreased even with the added load and store instructions.

3.2 The Compact Bytecode Set

The Squawk bytecodes are based on the standard Java bytecodes, with changes made to simplify execution and to save space. The complete Squawk bytecode set supports full CLDC functionality; a minimized subset has additionally been defined, with reduced operand field sizes appropriate for small devices (for example, with the minimized subset a class cannot have more than 256 static fields). This paper focuses on the minimized subset.

Many common bytecode sequences in standard Java require three bytes. In Squawk these sequences are reduced when possible to two bytes. For example, most branch instructions are two bytes in length, the branch bytecode being one and an 8-bit offset being the other.

The concept of the *wide* prefix in standard Java is enhanced in Squawk, so that an operand can be extended from 8 bits to 12, 16, or 32 bits. Returning to the branch example, occasionally the branch offset will exceed the 8-bit offset field and a wide prefix will be needed. Often a further 4-bits will be sufficient and this can be supplied using one of sixteen 4-bit wide prefix bytecodes. Thus a 12-bit offset is realized using a three byte sequence. In the rare case that this is insufficient, the offset can be extended to 16 or 32 bits using two more bytecodes called *wide_half* and *wide_full*. In these two cases the example branch would be 4 and 6 bytes in length respectively.

Local variables are explicitly typed in Squawk, eliminating the need for the type specific load and store bytecodes of the standard bytecode set. These bytecodes have been replaced with more load and store instructions that include the local variable number as part of the bytecode, increasing the number of such bytecodes from four each for loads and stores to sixteen. Many methods have more than four local variables, but relatively few have more than sixteen.

3.3 Bytecode Resolution

The standard bytecodes reference class members (fields and methods) symbolically via a name string rather than with an index or pointer. These symbolic references must be resolved sooner or later through some lookup process. In many Java interpreters this is done as the bytecodes are executed, and a common optimization is to then patch the bytecode stream with special resolved bytecodes that are often termed *quick* or *fast* bytecodes. This optimization is not suitable for systems that execute bytecodes in read-only or slow-to-write memory. Instead, in the Squawk system the bytecodes are resolved, as much as possible, when they are written into the memory of the target device during loading. At this time a field access can be resolved to an absolute offset within an object, and a method invocation can be resolved to a offset within a vtable. As a result the Squawk system does not need the symbolic constant pool found in standard Java systems, which saves a great deal of space.

3.4 Bytecode Verification

The restrictions placed on code structure, described above, make possible a simple two-pass verification process. The first pass reads the bytecodes one by one from an input stream, verifies that they are correct in terms of type safety, resolves class member references as described in the previous section, and writes them into their final location in NVM. A second pass is required to check that all branch targets are valid. The main advantages of this process are that only a few bytes of a method need be in RAM at one time, and that NVM (typically EEPROM) is written serially.

3.5 Object Construction

The standard protocol for constructing a Java object considerably complicates verification: the result of the *new* bytecode is an uninitialized object that must be regarded as a different type. This in turn entails creating a separate type for each `new` bytecode executed in each method. The overhead of maintaining these extra types is simply too high for Squawk, given the small amount of RAM available.

Squawk therefore handles object construction differently. Figure 1 shows a simple Java expression. When compiled with `javac` this expression becomes the bytecode sequence shown in Figure 2. This shows the new bytecode being used to create a prototype object. A copy of this object is made on the stack and the object constructor is called with this parameter. When the constructor returns the copy of the uninitialized object (which is now initialized) is stored into a local variable.

```
Integer one = new Integer(1);
```

Figure 1. An example of creating an Integer object

```
new java.lang.Integer
dup
iconst_1
invokespecial <init>(int)
astore_3
```

Figure 2. The corresponding bytecode for Figure 1

Our solution to the verification problem is not to expose uninitialized object references to the bytecode environment. Instead of having the new bytecode create the object, the object constructor method can do this, and return the address of the newly created object. Because object constructors are also called from other object constructors, a null object pointer must be passed to the outermost constructor to indicate this case, acting as a flag to indicate that a real object must be created. Constructors further up the class hierarchy must also be called and will be passed the reference to the original object in place of the null. Thus a constructor will allocate an object and replace the first parameter only if that parameter is a null.

Object construction is complicated by the fact that it must cause the initialization of the class of the object being created if it is not yet initialized. One of the less intuitive parts of the Java platform is that an expression such as:

```
new Foo(new Bar());
```

causes the `Foo` class to be initialized before the `Bar` class, yet the `Bar` object is created before the `Foo` object. It is essential that Java semantics be preserved in this respect. In order to satisfy the requirement that class initialization occurs at the correct time, the Squawk system uses the new *clinit* bytecode. This bytecode initializes the class as necessary.

Figure 3 shows the Squawk bytecode sequence for the example in Figure 1. The `Integer` class is initialized if necessary, and then the constructor is invoked (with a null receiver). This method returns the newly created object, which is then stored in a local variable.

```
clinit java.lang.Integer
const_null
iconst_1
invokeinit <init>(int)
store_3
```

Figure 3. The Squawk bytecode sequence for Figure 1

This different way of handling object construction does in fact lead to one small difference in the semantics of execution with respect to the Java standard. Out-of-memory errors can occur at slightly different times. In the standard case, where the object is allocated before the constructor is called, the error is thrown before the parameters for the constructor are evaluated. In the Squawk system execution occurs the other way around, so the evaluation of the parameters for the constructor can cause the memory error to be thrown before the object is allocated. Although this is a clear difference at the virtual machine code level it is less clear from the language specification that there is a correct order of allocation. This is made less clear still when one considers that calling the constructor may also fail due to inability to allocate an activation record.

3.6 The Format of Suites

The suite was specifically designed so that it could be read serially, with the information in the best order for installation using very little temporary (RAM) memory. The primary difference between a suite and a classfile is that in a suite all the class metadata for all classes comes before any of the methods for those classes. This means that by the time the bytecodes need to be verified all the class definitions have been processed.

A suite defines a collection of classes. The classes defined in the suite are called *Real Classes*. Classes external to the suite but used by classes in it are represented by *Proxy Classes*. Proxy classes contain much of the symbolic information found in the constant pool of a standard classfile. Proxy classes allow the fields and methods of classes external to the suite to be treated in the same way as the real classes defined in the suite.

3.7 Memory Issues

Getting Squawk to run on devices with small, tripartite memories raised some issues with respect to minimizing memory and placing data

3.7.1 Inter-region pointers

Squawk supports separate memories in three areas, RAM, NVM, and ROM. Objects in less permanent memory are allowed to contain references to objects in more permanent memory but not vice versa. Thus objects in NVM can refer to objects in ROM and objects in RAM can refer to objects in NVM or ROM.

3.7.2 The stack

The Squawk system uses a *chunky stack*, a stack composed of chunks allocated in the RAM heap and linked together. Each chunk is a Java object and thus can be garbage collected. Activation records are allocated from within a chunk.

3.7.3 Class definitions

Class definition information is divided into two types, immutable and mutable. For example, method bytecode arrays are immutable, while static variables are mutable. All objects have a one-word header that points to the immutable class information for that object type. The mutable state is held in an associative memory in RAM that is addressed using an internal class number. The bulk of the class definition is thus stored in NVM or ROM.

3.7.4 Monitors

Object monitors are used to implement synchronization. The monitors are placed in a LRU queue in RAM with the most recently used monitor at the head of the queue. All monitors hold a reference to their corresponding object. The garbage collector keeps monitors for live but unlocked objects (that is, the monitors are not currently being used) in the queue in case they are needed in the future. The act of allocating a monitor will cull unused monitors from the queue in order to prevent searching the queue from being a performance issue. Currently only six unused monitors are allowed in the queue.

4. IMPLEMENTATION

The Squawk system is implemented almost entirely in the Java language. As much as possible of it is implemented in standard Java code that is executed by the interpreter itself. This makes the system easy to develop and maintain. Such features as thread management, class initialization, object synchronization, and class loading are implemented this way. The remaining parts of the VM -- the interpreter, garbage collector, and native methods – are almost completely written in a Java subset that is also a subset of C. The VM thus is tested as a Java program before being compiled and debugged as a C program. The intersection of the Java and C languages is not feature rich; nonetheless, despite lacking such features as C structures and Java objects for describing runtime data structures, the interpreter is compact and readable. There are a few unavoidable syntactic differences between Java and C. A very simple Java-to-C converter (written in Java) performs these minor syntactic translations.

4.1 System data structures

All the data structures in the VM are implemented as Java objects, or as objects with a format compatible with the memory management system (stack chunks, for example, are implemented as a special form of integer array).

4.1.1 Methods

Methods are implemented as byte arrays. The first few bytes of these arrays contain some specially encoded method data that eliminates the need for a method object, which saves a significant

amount of space. The method header includes an object pointer map for the method's activation record (used by the garbage collector) and an exception handler table. Each class object has a pair of vtables that are used for the static and virtual methods for the class. These are normal Java arrays of byte arrays (byte[][]). Interfaces are handled using lookup tables that translate a method offset defined within an interface to a method offset in the virtual methods of the class.

4.1.2 Object pointer maps

Every class contains a reference to a byte array that contains an object pointer map used by the garbage collector to identify pointers in instances of the class. The translator, when it defines the internal representation of an instance, makes sure that pointer fields appear first. This canonicalization of representations greatly reduces the number of object pointer maps. Squawk has six predefined pointer maps that cover 92% (110 of 120) of all the classes in the CLDC runtime library.

4.1.3 Class state and initialization records

A class state record is the mutable part of a class definition. It contains the object reference fields and the integer fields used to hold the static variables of the class. The initialization state of the class is maintained in a separate data structure (also in RAM). A class can be in one of four states:

- It is not initialized. It has neither a class state record nor a class initialization record.
- It is being initialized. It has an initialization record.
- It is initialized. It has a state record.
- It cannot be initialized. It has an initialization record indicating this.

The class state records are associatively referenced and are kept in an LRU queue, which reduces access time.

The class initialization code is written in Java and is an exact transcription of the procedure described on page 53 of the *Java Virtual Machine Specification, Second Edition*[3]. The code uses the standard Java synchronization and object notification mechanisms. An optimization is employed to avoid this procedure in some cases. When a class has no `<clinit>` method and none of its super classes have one then the class is defined as not requiring initialization. Classes of this kind are immediately marked as initialized when initialization is required. This optimization is used when the system is started, so that the classes `java.lang.Class` and `java.lang.Thread` and the Monitor class do not require full initialization.

4.1.4 Threads, Monitors, and Activation records

Threads are implemented using a `java.lang.Thread` object and a logical chain of activation records. The thread object points to the youngest stack chunk when the thread is not running, and the stack chunk itself has a pointer to the youngest activation record it contains. Activation records are not objects in the heap because if they were the frequency of their allocation and deallocation would make the system slow. Instead, activation records are areas within a chunk and are allocated by moving a stack pointer within the chunk. When a stack chunk becomes too full to hold another activation record a new chunk is chained onto the old one and the new activation record is placed there. Old chunks are retained so that they may be reused. This can be a significant performance benefit in the case where a leaf routine is called frequently in a tight loop and its activation record cannot be allocated in the current stack chunk. The garbage collector must check chunks to see if they are truly unused (as it must also check unused monitor objects).

4.2 Garbage collection

The transformer makes garbage collection relatively simple. Objects and activation records have simple object pointer maps that identify object references, which make exact garbage collection possible.

Squawk guarantees that there are no C local variables in (the C version of) the interpreter by not invoking the garbage collector from the interpreter. Instead, the main control loop of the system is:

```
for(;;) {
        interpret(result);
        result = gc();
}
```

The interpreter only exits when memory allocation fails. The amount of memory required is recorded in a global variable when this occurs. When the collector is finished it compares the available memory with the amount needed, and returns false if there is still not enough. This flag is passed to the interpreter, which will then throw an `OutOfMemoryError` if the collector could not recover enough memory for the program.

This simple system works because the entire state of the virtual machine is represented using only objects in the object memory. There are certain special roots in the object memory, but these are known to the collector and interpreter and are referenced through accessor methods.

There are only three events in the interpreter that can cause a garbage collection: explicit invocation of the method `System.gc()`, the failure to allocate a stack chunk, and the failure to allocate an object. All three of these events occur on method entry. This is true for stack chunk allocation because when a method starts running it attempts to extend a preallocated minimal activation record to include the memory needed for its local variables and evaluation stack. It is true for object allocation because that is done as a side effect of entering an object constructor.

4.3 Thread scheduling

Thread scheduling is done in the class `java.lang.Thread`. A very small native method called from this class causes the virtual machine to stop running one thread and start running another one. Thus the thread scheduling code is factored out of the core VM, which makes it easy to change the scheduling algorithm. Thread preemption is achieved by decrementing a counter every time the interpreter executes a branch. When the counter reaches zero the interpreter automatically invokes the method `Thread.yield()`, which allows another thread to execute.

It is possible for the virtual machine to run out of active threads. This will occur when all the live threads are either waiting for I/O to complete or for a thread sleep operation to terminate. In this case a native method is called that causes the virtual machine to wait for a specified time or wait until the completion of an I/O

operation. When this native method returns the VM will find a thread to execute.

4.4 Native Methods and I/O

Native code that is used for I/O is modularized into components called channels. A channel has an API that must be adhered to in order for blocking I/O operations to correctly interoperate with the thread scheduling system.

The main principle by which the I/O system operates is that thread switching cannot occur while executing native code. This is an obvious consequence of the way thread scheduling is done (by Java code in the thread class). There is however a mechanism by which thread switching can occur when a long I/O operation takes place. This relies on close cooperation between the Java code and native code. When a channel is asked to perform I/O it can return an event number. The thread must then be blocked until the event occurs; there is a method in the thread library that does this. When the I/O event completes the native code will record the completed event number in a queue. The Java scheduling code in the thread library will periodically examine this queue and restart the blocked thread.

5. RESULTS

A prototype implementation of the Squawk system is complete, including the translator and the virtual machine. Enough Java API library support has been written to support the CLDC TCK compatibility tests and a few demos.

All 4628 TCK tests have been run on the system, and 4537 (98%) pass. The translator fails 43 due to the inability to handle esoteric or border case constructs. The VM fails 37 because of limits imposed by the minimized bytecode set, and the system fails 11 due to the current absence of complete runtime access control (the verifier currently ignores *private*, *protected*, and *public* access modifiers).

The static footprint of the core system, compiled from C, which includes the interpreter, a RAM garbage collector, and an NVM garbage collector, is 25918 bytes of x86 instructions (this does not include C libraries).

The minimum runtime footprint in RAM (the memory needed for the null program) is 520 bytes for the Java heap and 532 bytes for native stack and data (on the x86).

The runtime performance of the Squawk system is close to that of the KVM, ranging from 84% to 107% of the KVM for four benchmark programs.

Table 1. Execution times for Squawk and KVM

	Squawk	KVM	Squawk/KVM
delta blue	2864	2624	0.92
mpeg	8282	7020	0.84
cubes	4927	4226	0.86
hanoi	3556	3805	1.07

Note: times are in milliseconds; delta blue is a constraint based equation solver; mpeg is an mpeg decoder program, cubes is a 3-D rotation and display program; and hanoi is the tower of Hanoi program.

Both the KVM and Squawk interpreters are implemented as a switch statement in a loop written in C. The most likely reason for the Squawk system's slightly slower performance is that a number of bytecodes are implemented with Java functions (instanceof, checkcast, and monitor operations, for example), and although they are not being called often they are enough to cause these small reductions in performance.

Uncompressed, suites are on average 38% the size of JAR files. Compressed, they are 32%. (Uncompressed JARs are classes run through the JAR tool without compression, uncompressed suites are as emitted by the translator, compressed JARs are classes run through the JAR tool with compression, and compressed suites are suites that have been run through the JAR tool with compression.)

Table 2. Comparison of Uncompressed JAR and Suite Sizes

uncompressed	JAR	suite	suite/JAR
CLDC	458291	149542	0.33
cubes	38904	16687	0.42
hanoi	1805	835	0.46
delta blue	30623	8144	0.27
mpeg	100917	54888	0.54
manyballs	12017	6100	0.51
pong	17993	7567	0.42
spaceinvaders	50854	25953	0.51
tilepuzzle	18516	7438	0.40
wormgame	23985	9131	0.38
total	753905	286285	0.38

Table 3. Comparison of Compressed JAR and Suite Sizes

compressed	JAR	suite	suite/JAR
CLDC	238570	66109	0.28
cubes	21087	8548	0.40
hanoi	1259	934	0.74
delta blue	17011	4306	0.25
mpeg	42716	16713	0.39
manyballs	6786	3485	0.51
pong	9789	4120	0.42
spaceinvaders	21854	9064	0.41
tilepuzzle	10107	4124	0.41
wormgame	13253	4738	0.36
total	382432	122141	0.32

Note: manyballs is a simple thread demo; tilepuzzle and wormgame are simple games; and pong and spaceinvaders are Java versions of the popular games.

The hanoi benchmark is smaller as an uncompressed suite than as a compressed one because the suite version is small and already compact, so that running it through the JAR tool simply adds the overhead of the JAR file structure.

Leaving aside hanoi, the variance in size relative to the JAR format is explained by the ratio of large methods to small ones.

The per-method overhead is larger in a classfile than a suite, with the result that suites with many small methods will compare more favorably to their corresponding JAR files than suites with a few big methods.

Table 4. Comparison of Suite and NVM Image Sizes

	suite	image
cubes	16687	16400
hanoi	835	576
delta blue	8144	10152
mpeg	100917	52896
manyballs	6100	3700
pong	7567	5524
spaceinvaders	25953	22116
tilepuzzle	7438	7344
wormgame	9131	6952

Table 4 shows a comparison of the size of applications in their uncompressed suite formats versus their in-memory image formats. The sizes are all smaller for the image format except for delta blue. This is due to the way that String objects are currently set up in an image: they are converted from UTF8 encoding in the suite to 16-bit unicode encoding in the image. The delta blue suite contains a number of strings. An optimization is planned that will represent strings as byte arrays, which will solve this problem.

The above results are preliminary. A number of issues remain relating to the generation of translated bytecodes. For example, there is no "dup" bytecode in Squawk; it must instead be emulated using an additional local variable. As a result the translator does not always generate the best Squawk code. For example:

```
count++
```

is compiled by javac into the following 6 Java bytecodes:

```
aload_0
dup
getfield #17 <Field int count>
iconst_1
iadd
putfield #17 <Field int count>
```

which in turn is translated into 10 Squawk bytecodes:

```
load_0
store_5
load_5
getfield .ubyte 0
const_1
iadd
store_6
load_5
load_6
putfield .ubyte 0
```

In contrast, the logical equivalent:

```
count = count + 1
```

compiles to the following 6 Java bytecodes:

```
aload_0
aload_0
getfield #17 <Field int count>
iconst_1
iadd
putfield #17 <Field int count>
```

which in turn is translated into 4 Squawk bytecodes:

```
this_getfield .ubyte 0
const_1
iadd
this_putfield .ubyte 0
```

6. CONCLUSIONS AND FUTURE WORK

The Squawk system is a CLDC-compliant Java implementation for small devices. Preliminary results show it to achieve classfile compaction roughly comparable to other techniques. It does this, however, with considerably smaller on-device memory requirements, both in terms of the static system footprint and the RAM required to verify, load, and run applications. It does this at no speed penalty, compared to a reference interpreter.

It was successfully implemented completely in the Java language, including the core interpreter and garbage collector. Key components have been automatically converted to C and compiled, and the system runs natively on the x86 and SPARC platforms.

Anticipated future work includes: various improvements that will decrease size (addressing such issues as the string and dup problems) and increase speed; developing functionality related specifically to the Java Card platform (supporting, for example, NVM object storage and transactions); and porting the system to interesting small devices.

7. REFERENCES

[1] Alpern, B., et alia; "The Jalapeno virtual machine", IBM Systems Journal, Vol. 39, No. 1, 2000, pp. 211-238.

[2] Clausen, L.R., Schultz, U.P., Consel, C., Muller, G.; "Java Bytecode Compression for Low-End Embedded Systems"; *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 3, May 2000, pp. 471-489.

[3] Lindholm, T, Yellin, F; *The Java Virtual Machine Specification, Second Edition*; Addison-Wesley, April 1999.

[4] Pugh, W.; "Compressing Java Class Files"; Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99), 1999, pp 247-258.

[5] Rayside, D., Mamas, E., Hons, E.; "Compact Java Binaries for Embedded Systems"; *Proceedings of the 9th NRC|IBM Centre for Advanced Studies Conference (CASCON '99)*, 1999, pp. 1-14.

[6] Taivalsaari, A; *Implementing a Java Virtual Machine in the Java Programming Language*; Sun Microsystems Laboratories Technical Report TR-98-64, March 1998.

[7] Taivalsaari, A, Bush, B, and Simon, D; *The Spotless System: Implementing a Java System for the Palm Connected Organizer*; Sun Microsystems Laboratories Technical Report TR-99-73, February 1999.

[8] Tip, F., Sweeney, P.F., Laffra, C., Eisma, A., Streeter, D.; "Practical Extraction Techniques for Java"; *ACM*

Transactions on Programming Languages and Systems, Vol. 24, No. 5, November 2002, pp. 625-666.

[9] *The Squawk System, Preliminary Draft Specification 2.1*; Sun Microsystems Laboratories; 16 September 2002.

[10] Connected Limited Device Configuration: http://java.sun.com/products/cldc/

[11] Java Card: http://wireless.java.sun.com/javacard/

[12] Squeak Smalltalk implementation: http://www.squeak.org

[13] Esmertec's JBED ME: http://www.esmertec.com/technology/articles.shtm

[14] Aplix's JBlend: http://www.aplixcorp.com/products/jblend.html.

[15] Jikes Research Virtual Machine: http://www.ibm.com/developerworks/oss/jikesrvm/

[16] joeq virtual machine: http://sourceforge.net/projects/joeq http://www.stanford.edu/~jwhaley/

[17] Waba programming platform: www.wabasoft.com.

[18] IBM's WebSphere Micro Environment: http://www.ibm.com/software/wireless/wme/