Darjeeling, A Feature-Rich VM for the Resource Poor

Niels Brouwers
Embedded Software Group
Delft University of Technology
niels.brouwers@gmail.com

Koen Langendoen
Embedded Software Group
Delft University of Technology
k.g.langendoen@tudelft.nl

Peter Corke
Autonomous Systems Laboratory
CSIRO ICT Centre
peter.corke@csiro.au

Abstract

The programming and retasking of sensor nodes could benefit greatly from the use of a virtual machine (VM) since byte code is compact, can be loaded on demand, and interpreted on a heterogeneous set of devices. The challenge is to ensure good programming tools and a small footprint for the virtual machine to meet the memory constraints of typical WSN platforms. To this end we propose Darjeeling, a virtual machine modelled after the Java VM and capable of executing a substantial subset of the Java language, but designed specifically to run on 8- and 16-bit microcontrollers with 2-10 KB of RAM.

The Darjeeling VM uses a 16- rather than a 32-bit architecture, which is more efficient on the targeted platforms. Darjeeling features a novel memory organisation with strict separation of reference from non-reference types which eliminates the need for run-time type inspection in the underlying compacting garbage collector. Darjeeling uses a linked stack model that provides light-weight threads, and supports synchronisation.

The VM has been implemented on three different platforms and was evaluated with micro benchmarks and a real-world application. The latter includes a pure Java implementation of the collection tree routing protocol conveniently programmed as a set of cooperating threads, and a reimplementation of an existing environmental monitoring application. The results show that Darjeeling is a viable solution for deploying large-scale heterogeneous sensor networks.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

Languages, Experimentation, Performance

Keywords

Java, Virtual Machines, Wireless Sensor Networks

1 Introduction

Despite the resource limitations of the microcontrollers used in typical sensor nodes there is considerable attraction in the idea of running a Virtual Machine (VM). A virtual machine enables portability by abstracting the underlying platform and presenting a standard programming interface across a range of target platforms. This is important in the WSN context as we are moving towards heterogeneous networks, by design (i.e. two-tier architectures), or simply as a consequence of long-running systems evolving over time (e.g., adding new nodes after some years deployment). The support for dynamic loading of VM byte code provides programmers with the flexibility to update or extend a running application. Again, the long lifetime of sensor networks make this a crucial feature as bug fixing and application-level retasking are inevitable when applications are deployed for more than a few weeks.

Another type of benefit that a virtual machine provides is the ease of programming that comes with it, or actually, the languages it supports. The Java Virtual Machine (JVM) for example provides programmers with the luxuries of an object oriented language, dynamic memory management and protection, threads, and exception handling, none of which are provided by the underlying hardware. In addition, the JVM provides a safe execution environment ensuring that errors are handled gracefully and cannot take the entire software stack down as in the case for execution on bare hardware. This enhances the robustness of the software executing in the field, and more importantly reduces the code development and testing efforts considerably.

Java generally provides quicker deployment and increased maintainability over C/C++ [5]. This helps bring down the total cost of ownership of a sensor network not only during initial development, but also post-deployment.

The benefits of virtual machines for sensor networks have been recognised by the research community [11, 13, 19, 20] and the general consensus is that the price to be paid, that is the loss in execution speed, is of minor importance for the majority of WSN applications performing simple monitoring tasks. However despite this, most applications today are still written in low-level programming languages like NesC for TinyOS. The reasons are, firstly, most implementations of virtual machines have compromised on the supported functionality to reduce the memory footprint so that it can run within the few KB of RAM provided by typical sensor node platforms. Secondly, the state-of-the-art virtual machines are all proprietary implementations which slows their acceptance by the research community as it hampers the integration of experimental sensors, protocols, and algorithms.

In this paper we present Darjeeling, a virtual machine modelled after the Java VM and capable of executing a substantial subset of the Java language, but designed specifically to run on 8- and 16-bit microcontrollers with 2-10 KB of RAM. Darjeeling is, to the best of our knowledge, the first publicly available *open source* Java VM for wireless sensor networks. It provides a rich set of (Java) features including light-weight threads, dynamic memory management (garbage collection), and exception handling, while at the same time being optimised for resource-poor targets. This is accomplished by the combination of a standard Java compiler, an off-line byte code analysis and transformation engine, a 16-bit architecture run-time with a tailor-made instruction set, and a novel memory organisation that strictly separates references from other types eliminating the need for costly runtime type inspection during garbage collection.

To demonstrate these advantages we have implemented the Darjeeling VM on three different platforms with different microcontrollers (ATmega128, MSP430), operating systems (TinyOS, Contiki, FOS), and radios (CC1000, CC2420, nRF905). The performance in terms of memory footprint and speed is evaluated with micro benchmarks, as well as a real-world monitoring application including a pure Java implementation of the Collection Tree Protocol (CTP) [9] conveniently programmed as a set of cooperating threads. These results show that Darjeeling is a viable solution for deploying large-scale, heterogeneous sensor networks.

This papers makes three novel research contributions:

- it describes the Darjeeling VM, the design considerations, and the novel memory organisation that is key to achieving a low memory footprint;
- it details the implementation aspects for the three target platforms involved, showing that Darjeeling is portable across a wide-range of resource-poor devices;
- it demonstrates the feasibility of the Darjeeling concept through an in-depth evaluation of its performance for both micro benchmarks and a real-world environmental data collection application.

The remainder of this paper is structured as follows. The related work is surveyed in Section 2, followed by the design consideration of the Darjeeling VM in Section 3. Next, the implementation details regarding the linking model, memory organisation, byte code analysis, and execution model are presented in Section 4. Section 5 presents the microbenchmark performance evaluation for the Darjeeling VM on three target platforms, and Section 6 presents the evaluation of the CTP implementation and the re-implementation of a real-world monitoring application. Our conclusions are presented in Section 7.

Darjeeling is freely available from `http://darjeeling.sourceforge.net/`.

2 Related Work

Several VM implementations have been reported in the wireless sensor networks research community, and each strikes a different balance between flexibility, supported features, and resource (memory) usage. Some of the VMs are open source projects (e.g., Maté [13] and leJOS [21]), allowing other researchers to repeat experiments, use them for reference, and even extend them, while others are completely proprietary (e.g., Sentilla [19] and Java Card [2]). In general we can distinguish two competing philosophies: application-specific vs. generic VMs, which we discuss next.

Application-Specific Virtual Machines (ASVM) [14] are optimised for a specific problem domain and abstract common operations as instructions in a virtual machine. Programs tend to be very small, in the order of ten to a hundred bytes, which makes reprogramming nodes in a network very energy efficient. Examples of ASVMs are Maté [13] and VMSCRIPT [18]. VM* [11] is a Java VM project that advocates synthesis of VMs tailored for specific *applications*. It supports incremental linking to extend the VM as new features are needed, which overcomes the limited flexibility of typical ASVMs. Unfortunately VM* is closed source.

The class of generic VMs for sensor networks supports execution of some higher-level programming language, usually Java or a subset. This approach provides greater flexibility, for example to support application-level retasking, at a cost of larger program sizes and more complex VMs. In this context the Connected Limited Devices Configuration (CLDC) specification [1] is relevant as it describes a minimal standard Java platform for small, resource-constrained devices that have some form of wireless connectivity. Although often understood as describing cell phones, lately the specification has been increasingly mentioned in the context of sensor networks. The minimal CLDC hardware requirements are a 16- or 32-bit processor with 160 KB to 512 KB of memory available to the JVM, which is almost two orders of magnitude greater than what a typical sensor node provide.

Sun Microsystems has introduced several technologies related to Java on embedded devices. The Squawk project [20] provides an open-source, CLDC-compatible virtual machine, largely written in the Java language. The project introduces the concept of a *split VM architecture* where class loading and verification are done off-line, resulting in compact executables and low memory footprints. The Squawk VM runs on top of the Sun SPOT platform, an ARM-based sensor node with 512 KB of RAM and 4 MB of flash. This makes Squawk too large for our targeted platforms.

The Java Card [2] virtual machine targets 16-bit microcontrollers with approximately 2 KB of RAM. Although it does not support features such as multi threading or even garbage collection, it is significant for proposing a 16-bit architecture and a modified instruction set to execute Java programs more efficiently on micro controller platforms.

A recent generic VM is the leJOS [21] project, which features an open-source JVM that can execute on memory constrained devices, and has been demonstrated for wireless sensor networks in [7]. Unfortunately it lacks a number of essential features including garbage collection, so it is not suitable for more complex applications.

3 Design

Having identified the gap for an open-source feature-rich virtual machine for resource-poor platforms like sensor nodes, we have designed a new virtual machine called the Darjeeling Virtual Machine (DVM) that can execute a considerable subset of the Java language. Darjeeling does not (yet) support Java constructs such as floating point and 64-bit datatypes, reflection, and synchronised method calls, see Section 4.6 for details. Our virtual machine has been designed from the ground up for devices with small heaps and minimises memory consumption wherever possible.

3.1 Requirements

The primary design goal for Darjeeling is to provide execution of complex Java applications on a range of different sensor network architectures. This requires memory efficiency, portability, and some means to load and unload libraries and applications.

3.1.1 Memory Efficiency

In order to allow execution of meaningful applications Darjeeling should be memory efficient, as the small heap sizes of our target platforms are the main factor in constraining program complexity.

Threads especially should have as little overhead as possible, so that preemptive threading can be used freely by application programmers. Since we also want to allow multiple applications (each comprising a number of threads) running concurrently on a single node there must be little per-application overhead.

3.1.2 Portability

A virtual machine is middleware that sits in between the operating system (if any) and running applications. Many such operating systems exist for wireless sensor networks that provide different concurrency models such as event-driven or thread oriented. There are also different MCUs to deal with, most notably the ATmega and MSP430 series, with different memory types and sizes, and different addressing schemes. Therefore Darjeeling must be portable across all these MCUs and operating systems.

3.1.3 Application Loading

There are many cases in which it is useful to reprogram nodes after they have been deployed in the field. This can be to fix bugs, introduce new functionality, or completely re-task the network. While some testbeds allow reprogramming directly through wired means it is usually more practical to support application loading over the air.

A virtual machine can allow multiple applications to co-exist safely on a single node. Replacing separate applications is much more efficient than reprogramming an entire image. Darjeeling must therefore allow the loading and unloading, and starting and stopping of applications without having to reset the node or otherwise affect its running state.

3.2 Motivation for a New VM

The Java Virtual Machine (JVM) [16] is designed around 32-bit architectures with typically megabytes of memory. Certain design decisions that make sense for such platforms might not be ideal in the context of sensor networks where common MCUs use either 8- or 16-bit architectures and memory is measured in kilobytes.

3.2.1 Linking Model

The first issue we address is that of the Java class file format and the dynamic linking model. Java classes are linked dynamically with a per-class granularity. This allows for great flexibility, but comes at a significant cost. Java class files are generally large as they contain linking information in the form of string literals. A second problem with the dynamic loading technique is that it requires some linking information to be kept in RAM at run-time, introducing a non-trivial memory overhead for each loaded class.

Embedded JVMs commonly address this issue by employing a split VM architecture [20], and implement conversion tools that perform static linking between groups of class files to either reduce or completely remove the need for string literals in their respective file formats. This greatly reduces code footprint and eliminates the per-class memory overhead [4, 11, 17, 20]. The trade off is the loss of reflection and some flexibility in the linking model. This seems reasonable, especially as it allows for an efficient implementation where the byte code and class definitions can be kept in flash memory. Therefore Darjeeling uses a tool called the *infuser* that performs static linking of groups of class files.

3.2.2 Stack Width

In traditional Java VMs values are stored in 32-bit slots. This is true for the operand stack, local- and global variables, and inside objects. It allows for quick access on 32-bit architectures, but is impractical for memory-constrained 8- and 16-bit platforms. On these platforms references are typically 16 bits wide so storing them in 32-bit slots results in a 100% memory overhead. This applies also to smaller integer types such as `byte`, `boolean` and `short` which are commonly used in sensor network applications.

The Java Card virtual machine [17] addresses the memory overheads for narrow hardware platforms by using a more suitable 16-bit slot width. This requires a modified instruction set because Java automatically widens smaller integer types to a 32-bit `int` and does not contain instructions to modify 16-bit values directly. The 16-bit architecture also requires byte code analysis to optimise `int` based arithmetic to `short` arithmetic where this is possible. We have chosen to follow Java Card and use a 16-bit architecture. Arithmetic expressions are optimised by the Infuser tool.

3.2.3 Garbage Compaction

Frequent allocation and deallocation of objects causes the heap to become fragmented with holes that are too small to accommodate common allocation requests, essentially wasting space. This is highly undesirable on our memory-constrained target platforms, so a compacting garbage collection algorithm is required.

Compacting garbage collectors first mark all objects in use, then *slide* them to one side of the heap, eliminating any holes. After compaction, references to relocated objects have to be updated in the running state of the program. In order for this to work it must be possible to identify which elements of the running states are references, and which ones are not so that no integer values are modified by mistake. It is possible to include type information about class fields and global variables in the executable file. Types of values on the operand stack and in local variables, however, may change

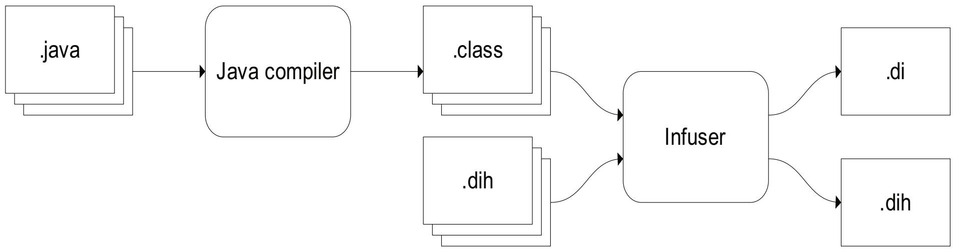

Figure 1. Infusion process.

dynamically so some mechanism must be in place to determine the types of these elements at run time.

One option is to simply type the stack and local variables as the program runs. Additional type information is kept on the stack and book-keeping is done when slots are accessed with `push`, `pop`, `load`, and `store` primitives. This method, called 'type tagging', is elegant in neither space nor time. An improvement is to do type inspection as the mark phase starts. The byte code is annotated post-compile with typing information called *stack maps* [3] at selected addresses. The VM can then infer the types of the stack elements at the current address in relatively few iterations from the closest precalculated state. This method is widely used but has the drawbacks that extra typing information must be added, which increases the code footprint, and that the type inference mechanism increases code complexity and reduces performance.

We have instead chosen an unconventional approach that strictly separates reference and non-reference types on the operand stack and in local variables. This solution comes at a cost of a single byte per activation record (see Section 4.2.2) and several extra instructions to manage the operand stack, but avoids any runtime handling of type information. We also separate references from non-references in objects, which allows for uniform treatment of stack and heap and simplifies the garbage collector even further.

4 Implementation

We have shown above that meeting the portability, memory-efficiency, and application-level retasking goals required us to let go of the JVM specification. Consequently we have chosen an approach where we designed our Darjeeling VM (DVM) to be similar enough to allow execution of post-processed Java programs, but with unique features that contribute to a low memory footprint. In this section we detail the implementation aspects of the Darjeeling VM, which has been ported to three platforms with different microcontrollers (ATmega128, MSP430), operating systems (TinyOS, Contiki, FOS), and radios (CC1000, CC2420, nRF905).

4.1 Linking Model

The Darjeeling runtime, the virtual machine, is responsible for executing Java programs. These programs comprise components such as class- and method definitions, executable byte code blocks, and string literals. These building blocks, which we shall call *entities*, are traditionally stored in Java `class` files.

In the Java world, individual `class` files are treated much like small libraries that are loaded on demand at run time. Classes contain a so-called *constant pool* that contains linking information. Any entity that is referenced by the class has an entry in the constant pool. A `new` instruction for instance, carries an index into the constant pool where information about the class to be instantiated can be found.

Darjeeling uses a *split VM architecture* [20] where class loading, byte code verification, and transformation is done off-line by a tool called the *Infuser*. Multiple class files are linked statically into loadable modules called *infusions*, which cooperate to form running programs. Named references are replaced with a numbering scheme so that keeping a run-time constant pool in memory is no longer necessary. Infusions are typically libraries such as the base infusion containing the `java.lang` package, or applications such as the CTP routing protocol application discussed in Section 6. Infusions 'flatten' a hierarchy of Java classes into lists of entities, and can import other infusions and reference the entities therein.

The process is shown in Figure 1. A series of Java source files is fed into a standard Java compiler producing a corresponding set of class files. These class files are then input to the Infuser tool along with one or more infusion header files. A call to the infuser typically produces two files: a Darjeeling Infusion (.di) file, and a Darjeeling Infusion Header file (.dih). Informally speaking, the infusion file contains the class definitions and byte code, and the headers contain linking information. Together these two files form the infusion.

4.2 Memory Organisation

Memory management in Darjeeling was implemented with two goals in mind: memory efficiency and separation of reference and non-reference types. Our linked stack architecture described in Section 4.2.1 enables threads with low fixed overhead. Section 4.2.2 and 4.2.3 describe how reference and non-reference types are separated on the operand stack and in heap objects, respectively.

4.2.1 Linked Stack

Each method call produces a new stack frame (activation record) as a context for the execution of that method. Java stack frames contain a local variable section, various bookkeeping values, and an operand stack. Bookkeeping includes a return address, stack pointer, and various context variables such as a pointer to the method that is being executed.

Traditionally stack frames are allocated on a single, preallocated space. The local variable section of a frame is located at the start of a frame, so that the frame of the caller can be overlapped with the callee. The last *n* active slots (parameters) on the operand stack of the caller overlap with the

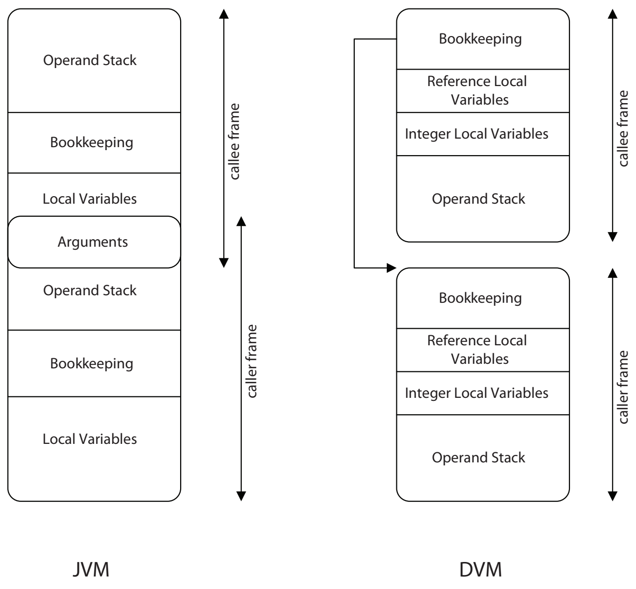

Figure 2. Stack frame layout comparison.

the first *n* slots (method arguments) of the callee. This allows for efficient passing of parameters between the two methods.

The drawback of the traditional method is that the size of this pre-allocated stack is equal to the worst-case requirement, introducing a severe memory overhead for each running thread. For Darjeeling we chose to implement *linked stack management* [22] where stack frames are allocated ad-hoc from the heap. This allows thread stacks to grow and shrink as the program runs, allowing for light-weight threads. An obvious drawback is that each stack frame requires a heap chunk header, but we note that the benefits of dynamic stack allocation greatly outweigh the cost of this overhead (see Section 5).

A comparison of the two layouts is shown in Figure 2. A typical JVM stack frame organisation is shown on the left, our linked stack model is shown on the right. Our stack frames consist of the same elements, but the local variable section is split into two parts for reference and non-reference types to help with garbage collection and compaction (see below). Since stack frames are allocated on the heap, and the local variables of our stack frames are split into two separate sections, it is impossible to directly overlap the operand stack of the caller with the the local variables of the callee. One option is to copy values from the operand stack of the caller to the local variables of the callee, but this is inefficient because it would duplicate values and waste space. Instead, we added specialised instructions to let the caller access the operand stack of the callee directly for retrieving arguments.

In order to successfully invoke a method there must be enough space on the heap to allocate an activation record. If a method cannot be called because the VM is out of memory a `StackOverflowError` is thrown. The programmer is responsible for catching this error and trying to recover, possibly by killing other threads or freeing up memory. If the error is uncaught the thread will be terminated (and its memory reclaimed).

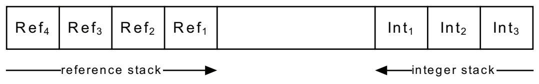

Figure 3. Stack organisation.

4.2.2 Double-Ended Stack

Darjeeling uses two operand stacks instead of one. One of the stacks holds the reference types, the other is for non-reference types. The stacks are allocated within the same memory space, the size of which can be easily obtained by byte code analysis. Two stack pointers are used; one is initialised to the lower bound of the stack space, the other to the upper bound. The first grows upwards, the other downwards. This is shown in Figure 3. When the root set needs to be marked, the collector can simply traverse the reference stack and directly mark each heap object it encounters. This reduces the complexity of the root set marking phase to $O(n)$ and eliminates any false positives. During compaction, every time an object is moved, the collector can traverse all the reference stacks and update the pointers in place.

The cost of this technique is that each stack frame now contains two stack pointers instead of one. In our stack frames these are stored using a single byte indicating the number of elements on each of the stacks, so that the overhead is just a single byte per stack frame.

In order for this technique to work some modification must be made to the byte code instruction set to replace stack manipulation instructions like `pop` with `ipop` and `apop` for integer and reference pop respectively. How this is accomplished is described in Section 4.5.

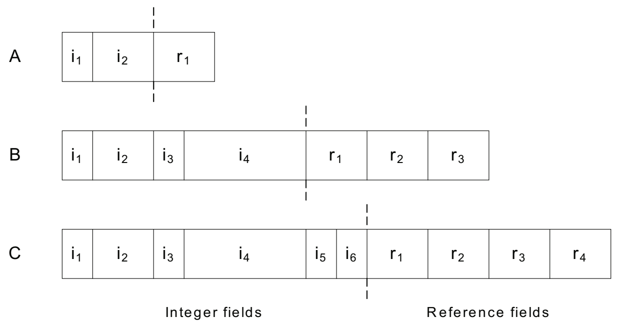

Figure 4. Java objects on the heap.

4.2.3 Objects

Strict separation of integer and reference types has also been applied to the layout of objects in memory. Figure 4 shows three objects allocated on the heap. The dotted line indicates the partition between the integer and reference fields. Darjeeling packs the integer fields of objects on the heap so that fields of type `byte` or `short` occupy only 1 or 2 bytes respectively instead of the standard 4.

The instructions `getfield` and `setfield` have been replaced by new instructions `getfield_<T>` and `setfield_<T>`, where *T* is one of `int`, `short`, `byte` or `ref`. The offset of the field inside the integer- or reference block is stored as an immediate value in the instruction. Since there are different `getfield` and `setfield` instructions for

Listing 1. The Darjeeling main loop on FOS.

```
while (dj_vm_countLiveThreads(vm)>0)
{
  dj_vm_schedule(vm);

  if (vm->currentThread!=NULL)
    dj_exec_run(RUNSIZE);

  sleep = dj_vm_getSleepTime(vm);
  if (sleep==0)
    fos_thread_yield();
  else
    fos_thread_sleep(sleep);
}
```

Listing 2. A native method implementation.

```
// javax.fos.Leds.setLed()
void javax_fos_Leds_void_setLed_byte_bool()
{
  // pop arguments off the stack
  // in reverse order
  int16_t on = dj_exec_stackPopShort();
  int16_t id = dj_exec_stackPopShort();

  // set the appropriate leds
  if (id==0) fos_leds_blue(on);
  if (id==1) fos_leds_green(on);
  if (id==2) fos_leds_red(on);
}
```

each data type the VM knows how many bytes to read or write at the given offset. Indexing into the reference fields is slightly more involved, as the offset of the reference field block must first be obtained. This is achieved by retrieving the size of the integer field block from the class definition which is stored in flash.

In our example *B* is a child class of *A*. It is always possible to substitute an instance of class *B* for an instance of its parent class *A*, as inheritance dictates, because the integer- and reference blocks are handled separately.

4.3 Execution

One of the benefits of Java programming over C-based middleware is that it provides an intuitive, preemptive multithreading concurrency model. Darjeeling implements preemptive multithreading in a very simple and straightforward way, allowing it to run on top of event-based, thread-based, and protothread-based concurrency models and even on systems that do not provide concurrency at all. In essence the running virtual machine is a polling loop that alternates between a `dj_vm_schedule()` and `dj_exec_run()` call. The former decides which thread should be run next, and the latter executes the specified number of atomic JVM instructions. At the moment Darjeeling does not support priorities and threads are simply scheduled in a round-robin fashion. This can be changed when needed, but the current approach of embedding time-critical code in native methods has proven itself a straightforward and effective alternative.

The time-slicing is not timer-based, although it is possible to call `dj_exec_run()` with a large number of instructions to execute and call `dj_exec_breakExecution()` on a timer to interrupt the thread after a fixed interval. Listing 1 shows how Darjeeling runs on top of FOS, a cooperative thread-based operating system for the Fleck wireless sensor module developed at CSIRO [6]. The `dj_vm_getSleepTime()` method returns the number of milliseconds until one of the threads has to be woken up. If there are one or more threads currently in the running state this method returns zero.

In case of the TinyOS port, if there are no threads to schedule immediately, a single-shot timer is used to renew the virtual machine task in the future. On FOS we use the API call `fos_thread_sleep()`. Both methods allow the underlying OS to put the MCU into low-power mode.

4.3.1 Native Code

In order for Java programs to access the node hardware, native libraries, and the virtual machine state, some mechanism for calling native methods is required. The Java language provides a `native` keyword for methods to signal that the implementation of that method is implemented natively rather than in Java. The infuser tool can generate C stubs from these Java method declarations for the application programmer to implement. These are then linked into the final image and programmed into the node.

Listing 2 shows the native implementation of the `javax.fleck.Leds.setLed()` method. Parameter passing is done through the operand stack, in reverse order. A native method may optionally return a value by pushing it onto the operand stack.

Some interaction with native hardware requires a Java thread to block until some event has occurred. A common example is waiting for a message from the MAC layer. On FOS the MAC `send()` and `receive()` methods simply block the calling thread. TinyOS uses split-phase interfaces where an event is generated when the send operation has been completed, and a receive event may occur at any time.

We expose these mechanisms using two stage locking. On the Java side a `synchronized` block is used to ensure that only one thread at a time may be blocked for an event from the underlying OS. If multiple threads call the same method they will be blocked waiting for the first thread rather than for the OS event. Listing 3 shows how this is implemented for the `receive()` method on the MAC layer.

The `receiveLock` monitor prevents multiple Java threads trying to process the same radio packet when a receive event occurs. A thread calling the native method `_receive` will be blocked immediately and another thread is scheduled for execution. When a receive event occurs the previously blocked thread is reactivated.

4.4 Infusion Management

Applications and libraries executed by Darjeeling are stored as infusions. Infusion files (.di) typically reside in program flash. When an infusion is loaded by the VM it will execute its class initialisers, and if the infusion has an entry point method, it will create a new thread and execute that method in the new thread. Although in traditional JVMs the class loading is lazy and determines the order in which

Listing 3. Locking on the Java side.

```
private static Object receiveLock =
        new Object();

// blocks until a message arrives
private static native byte[] _receive();

public static byte[] receive()
{
  synchronized(receiveLock)
  {
    return _receive();
  }
}
```

class initialisers are run, Darjeeling runs them in the order in which they appear in the file.

While loading new infusions is fairly trivial, unloading them is more involved since the VM must be left in a correct state after the infusion has been removed. Any references to elements from the infusion to be unloaded in the running state of the various threads may cause faulty behaviour when those elements can no longer be accessed.

Before an infusion may be unloaded any threads that are executing method implementations from that infusion must first be killed. This still leaves objects on the heap that are instances of classes defined in the unloaded infusion. These objects cannot be kept as they can no longer be accessed, their access methods have been unloaded, but they also can not be deallocated because there might be references to them from running threads. For example, when an application, passes an event handler to a routing protocol and later gets unloaded, the event handler object persists because the routing protocol still holds a reference to it.

We considered killing all threads that have direct or indirect references to such ‘bad’ objects but decided against it because it would not only involve implementing another marking algorithm, but also because in our example it would cause the routing protocol to be killed as well. We chose a solution that allows the application programmer to decide how to deal with such cases.

Instances of classes that are unloaded are marked ‘invalid’ by the unloading mechanism. Accessing these objects causes the VM to throw a `ClassUnloadedException`. This exception can be caught, allowing the application to remove the reference. If it is not caught the thread will terminate due to an uncaught exception. Either way, the VM is left in a correct and predictable state.

4.5 Byte Code Transformation

Darjeeling has a custom byte code instruction set that is optimised for 16-bit arithmetic, supports a double-ended stack architecture, and has typed versions of field get and set methods. It is the job of the infuser tool to map Java byte code to Darjeeling byte code. There is not always a one-to-one mapping between instructions and the transformation may be context sensitive. In these cases the generated Darjeeling instruction depends on the input types of the source Java instruction. To allow for these context sensitive transformations type inference has to be performed to determine the input types of each instruction. Figure 5 shows the transformation pipeline; the various stages are discussed next.

Listing 4. The max() function.

```
public static short max(short a, short b)
{
  return a>b?a:b;
}
```

Table 1. Java byte code for max().

	in	out	instruction	pre stack	post stack
0		1	iload_0		short
1	0	2	iload_1	short	short, short
2	1	3, 5	if_icmple(5)	short, short	
3	2	4	iload_0		short
4	3	6	goto(6)	short	short
5	2	6	iload_1		short
6	4, 5		ireturn	short	

4.5.1 *Import and Type Inference*

First the Java byte code is imported and translated into corresponding Darjeeling byte code. Any instructions that cannot be translated right away, such as stack manipulation operations, are replaced with place holders.

A type inference algorithm determines the states of the local variables and operand stack before and after each instruction. These states are called a handle’s pre- and post state. The post state is calculated from the pre state by simulating the effect the instruction has on the operand stack and local variables. The pre state of an instruction handle is derived by merging the post states of all its incoming handles.

Usually type inference is used for byte code verification and only the types of values on the operand stack are inferred. The infuser instead tracks which instruction handle produced each value and infers the type from there. This allows for multi-pass optimisation with type information being updated automatically when instructions are replaced.

Consider Listing 4, a function from the `Math` class that calculates the maximum of two shorts. The corresponding annotated Java byte code is shown in Table 1. The first column shows the offset of each instruction, the set of incoming and outgoing offsets (direct predecessors and direct successors) are shown in columns two and three. The pre- and post-stack columns show the *logical* types of the operand stack. We distinguish between logical types, the ‘actual’ type of the value, and the *physical* type. In the JVM `short` values are automatically widened to `int`, so although only shorts are used in our example their physical types are actually `int`.

The resulting Darjeeling byte code, after this phase, is shown in Table 2. The `iload` and `istore` instructions have been replaced by their short-sized counterparts, and since the method returns a value of type `short` the `ireturn` instruction has been replaced with `sreturn`. The `if_icmple` instruction, which compares two integer values, has not been optimised to its short form yet. This is done in the next phase.

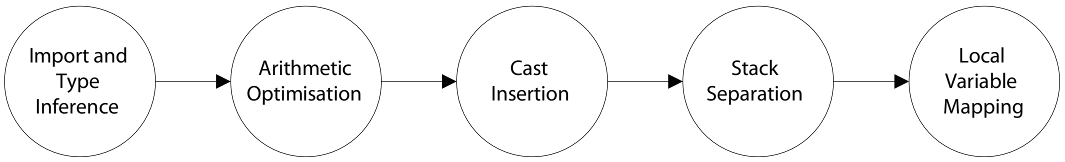

Figure 5. byte code transformation pipeline.

Table 2. Darjeeling byte code for max() after import

	in	out	instruction	pre stack	post stack
0		1	sload_0		short
1	0	2	sload_1	short	short, short
2	1	3, 5	if_icmple(5)	short, short	
3	2	4	sload_0		short
4	3	6	goto(6)	short	short
5	2	6	sload_1		short
6	4, 5		sreturn	short	

4.5.2 *Arithmetic Optimisation*

The JVM automatically widens small integer types to `int`, and all arithmetic is done using 32-bit intermediate values and instructions. To match the native integer size of most micro controller platforms Darjeeling can work with the `short` data type directly, widening values of type `boolean` and `byte` to 16 bits. The infuser optimises Java byte code to use short-typed instructions where this is possible, but must be careful when optimising arithmetic to make sure that expression semantics are preserved. The arithmetic optimisation stage performs this transformation.

Many Java instructions that operate on `int` types have short-typed counterparts in the DVM instruction set. The `iadd` instruction pops two 32-bit integer values off the operand stack, adds them, and pushes the result back onto the stack. It is important to realise that the result is implicitly truncated to 32-bit. Darjeeling has a short-sized version called `sadd` that performs the same calculation, only using 16-bit values. Because `sadd` truncates the result it cannot always be substituted for `iadd` even if both input types are `short`, because the addition might generate overflow that is relevant for later computation.

Our arithmetic optimisation algorithm determines which instruction sequences may be optimised by establishing which instructions potentially generate overflow, and whether that overflow should be preserved. For example, if the result of an addition is stored in a `short`-typed variable or is explicitly cast to `short` the potential overflow will be discarded anyway, so there is no need to keep it in the first place, and the cheaper `sadd` instruction can be safely used.

The algorithm starts by annotating the input byte code with two flags. The *gen* flag is set for every instruction handle that may generate overflow. The *keep* flag is set for instructions whose overflow should be preserved. An instruction that has both flags set cannot be safely optimised. The keep flag is generated by looking for instructions that require overflow to be preserved for one or more of their input values, and setting the keep flag on the instruction(s) that generate them.

Listing 5. Addition chain.

```
short a,b,c,d;
int e = a + b + c + d;
```

Consider the Java fragment in Listing 5. The corresponding annotated byte code is shown in figure 6. All the addition instructions have the gen flag set because they take short values as input, and this can generate overflow from `short` to `int`. The backwards pointing arrows show which instructions cause the gen flag to be set on which of their inputs. The `istore_4` instruction for instance requires any overflow generated by the last `iadd` to be preserved. In this state the algorithm is done immediately because there are no instructions that can be optimised.

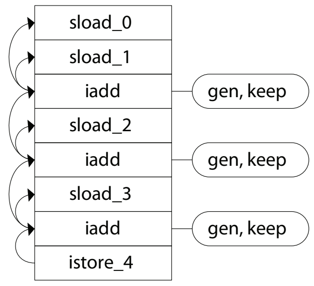

Figure 6. Annotated byte code.

Now consider a slightly altered version of the addition chain as shown in Listing 6. In this case the result of the addition is explicitly cast to `short`. The corresponding code is shown in Figure 7(a). Because the `i2s` instruction removes any overflow from the last `iadd` it no longer has the keep flag set. The algorithm can now safely replace it with an `sadd` because both input types are short and overflow is irrelevant.

Listing 6. Truncated addition.

```
short a,b,c,d;
short e = (short)(a + b + c + d);
```

Whenever an instruction is optimised the change may affect the keep flag on its inputs and gen flags on the instructions that use its output. The result of optimising the `iadd` instruction is shown in Figure 7(b). By applying this mechanism iteratively the entire expression can be optimised to use only short instructions, as shown in Figure 7(c) and 7(d).

4.5.3 *Cast Insertion*

The arithmetic optimisation stage cares only about logical types of values, not about the physical types in which they are wrapped. Recall the example in Figure 6, where the `iadd` instructions require `int` typed inputs. The `sload` instructions produce short values on the 16-bit stack, so these must be widened to 32-bit `ints` using an `s2i` instruction.

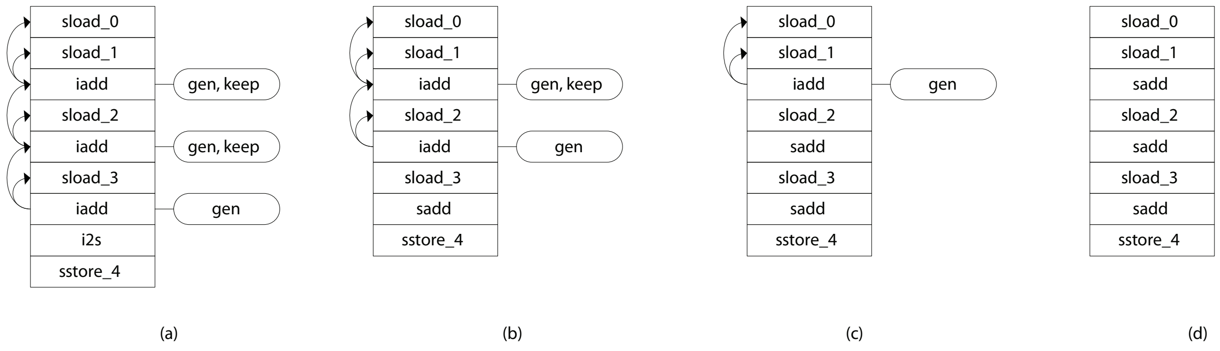

Figure 7. Arithmetic optimisation.

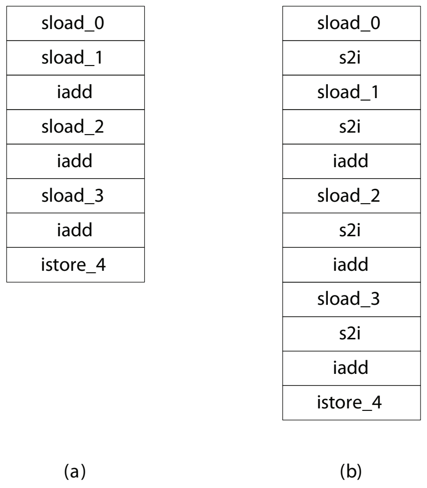

Figure 8. Cast insertion.

The byte code listings before and after the transformation are shown in Figure 8(a) and 8(b), respectively.

4.5.4 Stack Separation

As discussed in sections 3.2.3 and 4.2.2, Darjeeling uses a double-ended operand stack to eliminate the need for runtime type inspection. Logically speaking there are two stacks, one for integers, and one reference types. For most instructions it is immediately clear which stacks they mutate, but for some instructions this is context sensitive and type analysis is required for correct translation.

We illustrate the context sensitivity for the `dup_x1` instruction, whose effect is to 'duplicate the top element on the stack and insert it one place down'. This can be written as $..., v_2, v_1 \rightarrow ..., v_1, v_2, v_1$. The correct replacement for `dup_x1` depends on the values of v_1 and v_2. These can be any of `short`, `int`, or `ref`, yielding 9 different possible combinations. For simplicity we shall omit the `int` type, but it should be clear that the case can be easily extended to include other data types. The possible combinations of input types for the instruction are shown in Table 3. The three columns show the contents of the unified (Java) stack, the reference stack, and the integer stack respectively. From this table we can see how the transformation $..., v_2, v_1 \rightarrow ..., v_1, v_2, v_1$ would apply to each case. Table 4 shows the correct replacement instruction for `dup_x1` for each of the four cases.

Table 3. Stack contents at dup_x1.

Unified	Ref. stack	Int. stack
$v_2 : short, v_1 : short$	...	$v_2 : short, v_1 : short$
$v_2 : short, v_1 : ref$	$v_1 : ref$	$v_2 : short$
$v_2 : ref, v_1 : short$	$v_2 : ref$	$v_1 : short$
$v_2 : ref, v_1 : ref$	$v_2 : ref, v_1 : ref$	...

Table 4. Replacement instructions for dup_x1

Unified Stack	DVM instruction
$v_2 : short, v_1 : short$	`idup_x1`
$v_2 : short, v_1 : ref$	`adup`
$v_2 : ref, v_1 : short$	`idup`
$v_2 : ref, v_1 : ref$	`adup_x1`

If both v_1 and v_2 are of the same type, `dup_x1` can simply be replaced with a dup_x1 that operates on the appropriate stack, `idup_x1` for integers and `adup_x1` for references. If they are of different types, the top element and the element below it are located on different stacks, so the instruction can be replaced by an appropriate duplicate instruction.

4.5.5 Local Variable Mapping

Java compilers such as `javac` are usually not very efficient when it comes to assigning local variables to local variable slots because for most virtual machine implementations memory consumption due to stack space is not a major issue, and the assumption is made that frequently executed code will be jitted (compiled) anyway. Usually one or more slots are allocated for each local variable, leading to memory wastage when two or more variables can be mapped onto the same space. The infuser tool performs live range analysis to determine which variables can be mapped onto the same slot(s) and remaps the local variables accordingly. Darjeeling does not require the Java compiler to supply type information about local variables. Instead this information is inferred from byte code analysis so that even when a variable is declared `int` it may still be optimised to the `short` type.

4.6 Limitations

We have chosen to support a subset of the Java language. Like Java Card [17] we do not support floating point or 64-bit datatypes. We are looking at supporting the `long` type in the future to increase compatibility with CLDC 1.0.

The DVM does not support reflection since the required type information is not stored in our executable file format. The `synchronized` modifier on static methods is not supported because there is no loaded class to synchronise on, but it can be easily simulated using `synchronized` blocks.

5 Benchmarks

To assess the performance of the Darjeeling VM we created a set of benchmark programs that exercise specific parts of the implementation in terms of execution speed and code size. Performance in a real-world environmental monitoring application is discussed in Section 6.

5.1 Performance

The execution performance of the DVM was measured using three micro benchmarks. First we ran a bubble sort algorithm that sorts 256 32-bit integer values, initialised to strictly decreasing values. The second test is an 8×8 vector convolution that is performed 10,000 times, also using 32-bit values. Finally we hand-optimised an implementation of the well known MD5 hashing algorithm, generating MD5 hashes for the empty string, 'a', 'abc', 'darjeeling', and 'message digest'. These hashes are generated 1,000 times.

Each of the benchmark tests has both a C and a Java version and is written in such a way that the difference between both versions is almost completely syntactical. We were careful to eliminate any memory allocation after the test initialisation to make sure the garbage collector would not be triggered and skew the results. Also note that all tests ran with run time checks such as null pointer and bounds checking enabled.

Table 5. Performance benchmarks.

	C	Java	Java/C	instructions/s
Bubble sort	0.3 s	23.3 s	77.7	60,618
Vector conv.	9.1 s	496.7 s	54.47	65,454
MD5	13.1 s	399.7 s	30.4	52,294

We ran the tests on an ATmega128 microcontroller at 8 MHz, with all checks, such as bounds checking and null pointer checking, enabled. We tried running some tests with these checks disabled but didn't observe a significant speed-up. The results are shown in Table 5. The C and Java columns show the run times for the bubble sort, vector convolution, and MD5 tests respectively, and the Java/C column shows their ratio. The final column shows how many Java instructions per second were executed during each of the tests.

In these particular tests the performance penalty of the Darjeeling VM was about 30-78x. The measured byte code instruction throughput was between 52-65 K instructions per second. Note that although the VM was executing more instructions per second on the bubble sort test than on the MD5 test, the first is much slower compared to its native counterpart than the latter. We can explain this by looking at how VM instructions are interpreted.

Interpreting a VM instruction has a fetch, decode, and execute phase. While the fetch and decode components are similar for each instruction, there are large time differences for the execution phase. Method invocation and type checking instructions take longer to execute than simple arithmetic, and instructions that operate on 32-bit values are generally slower than their 16-bit counterparts. During the MD5 test the interpreter is executing fewer, but more complex instructions per second. The bubble sort test uses more trivial instructions, causing a greater relative overhead due to fetching and decoding.

Table 6. Performance benchmarks, 16-bit.

	C	Java	Java/C	instructions/s
Bubble sort	0.2 s	19.3 s	81.5	71,526
Vector conv.	4.6 s	520.9 s	113.2	71,512

We illustrate this effect by re-running the bubble sort and vector convolution tests, but this time altered to only use 16-bit data types (the MD5 algorithm is a 32-bit algorithm so could not be rewritten). The results are shown in Table 6.

The interpreter has a higher instruction throughput in both tests due to the simpler instructions, but the fixed overhead makes it unable to take advantage of the `short`-typed arithmetic as much as the native C implementations. The 16-bit Java version of the vector convolution test has even become slightly slower compared to the 32-bit one, which is due to the infuser not being able to completely optimise the arithmetic and having to insert explicit conversion instructions between the `int` and `short` data types (see Section 4.5.2). As expected, the interpreter is performing much worse on these new tests compared to native C, with measured performance penalties of 82x and 113x for the bubble sort and vector convolution tests respectively.

These results illustrate that the performance overhead of Darjeeling is difficult to measure because it is very much dependent on the application under test. A high instruction throughput does not necessarily mean a good performance, especially when compared to equivalent native code. We found that the cost of interpreting Java versus executing native C code, in terms of execution time, is a factor in the range of 30 to 113 times, or approximately two orders of magnitude.

5.2 Code Size

As discussed in Section 4.1, Darjeeling uses a split-VM architecture. This allows for considerably smaller executables. Table 7 compares the size of several infusions when stored as Java .jar files, which are zip compressed archives that contain .class files, and our .di file format which uses no compression but eliminates all string literals.

Table 7. File format comparison (sizes in bytes).

	jar	di	reduction
Blink	856	146	83 %
CTP	27,349	7,889	71 %
Test suite	42,834	19,023	55 %
Base library	14,795	3,591	75 %

Size reduction is related to the ratio of string literals to byte code in the input .class files. The base library for instance is mostly made up of class and interface definitions with little to no actual byte code, which accounts for the significant reduction in code size (75 %). The same holds for the exemplary TinyOS application, blink, which only has a

Table 8. Code size comparison (sizes in bytes).

	AVR	MSP 430	DVM	DVM (opt.)
MD5	11,364	9,396	4,513	4,364

single method implementation. The CTP routing protocol library (see Section 6), is also reduced considerably in size because most methods are short, just a few lines of code. This was done to improve modularity and readability in line with good software engineering practice. In contrast the test suite has relatively long methods each performing a series of short tests, which is why it was reduced less than the others.

The conclusion is that although writing modular code with many small methods, classes, and interfaces will still add to the code size, the cost is not nearly as high as with the traditional .jar file format because Darjeeling does not retain the string literals in the infusion file.

Proponents of virtual machines and stack-based instruction sets often quote the smaller code size as a major advantage. To measure this effect we compared code sizes of the MD5 hashing algorithm introduced in Section 5.1 for the native and interpreted cases. We compiled the C code for the AVR and MSP430 MCUs using the respective GCC ports at various optimisation levels (including `-Os`) and selected the smallest size, measured as the length of the `.text` segment, for each. We also compiled the equivalent Java code using the Sun JDK 1.6 compiler and noted the size in bytes of the resulting `.di` file. Finally, since most Java compilers do not aggressively size-optimise their output, we optimised the code using ProGuard version 4.4 [12], which reduced the final size of the Java code somewhat, see Table 8. Note that the MD5 code by itself is rather large, about 400 lines of C/-Java code, due to manual loop unrolling applied to the core algorithm to boost performance. In this test the AVR and MSP 430 native code sizes are 260% and 215% that of the optimised DVM case, respectively.

5.3 Stack Space

In order to measure the benefits of our 16-bit stack architecture in terms of memory usage during execution, we measured the average size of stack frames for methods in the four benchmark infusions. The size required for a single stack frame is the sum of the operand stack size, the size required for the local variables, and some fixed overhead for bookkeeping such as return address, stack pointers, and so forth. We calculated this number both for the 32-bit JVM case and our 16-bit DVM.

Table 9. Local variables and operand stack size (bytes).

	JVM	DVM	Reduction
Blink	10	6	40 %
CTP	11.4	6.3	45 %
Test suite	13.7	9.3	32 %
Base library	7.1	4.1	42 %

Table 9 shows the size of the variable part of the stack frame, the operand stack and local variables, for each of the test cases. During the development of the CTP library we used short types for loops counters and intermediate values wherever possible, resulting in a reduction of 45 %. The test suite on the other hand uses integer types in many places, but still stack usage is reduced by 32 %. This is mainly due to reference types being reduced from 32-bit to 16-bit. Table 10 shows the average *total* stack frame size, including the fixed overhead (8 bytes) due to the bookkeeping section. It shows that the total reduction is between 20-27 % for our benchmark infusions.

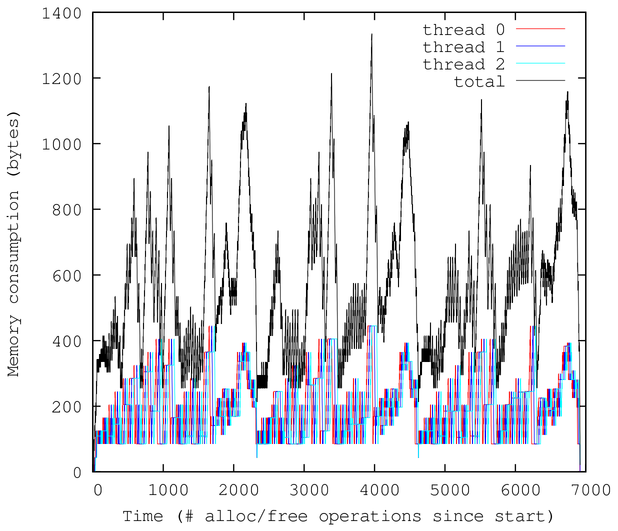

Figure 9. Garbage collection, unsynchronised.

Table 10. Total stack frame size (bytes).

	JVM	DVM	Reduction
Blink	18	14	22 %
CTP	19.4	14.3	27 %
Test suite	21.7	17.3	20 %
Base library	15.1	12.1	20 %

5.4 Multithreading

Our linked stack architecture allocates stack space on demand, allowing threads to grow and shrink their memory usage as they run. To illustrate how this affects memory consumption we ran a garbage collection test from our test suite. It sorts 20 numbers by inserting them into a binary tree, causing not only a large number of small linked objects to be created, but also generating a lot of nested method calls. The test calls the algorithm three times, in three concurrent threads.

Figure 9 shows the amount of stack space for each thread as the test ran, including the thread object and heap manager headers. Each thread uses up to 445 bytes, which makes the total peak at 1335. This means that the worst-case total stack usage for this test is equal to the sum of the worst-case values of the individual threads.

We can decrease the memory usage by allowing only one of the threads to perform the tree sort algorithm at a time. We ran the test again, but this time we added a `synchronized` block around the call to the algorithm, essentially serialising individual runs. The trace for this second run is shown in Figure 10. This time the total peaks at a mere 535 bytes, which is the peak value of a single thread, 445 bytes, plus the overhead of the two blocked threads at 45 bytes each.

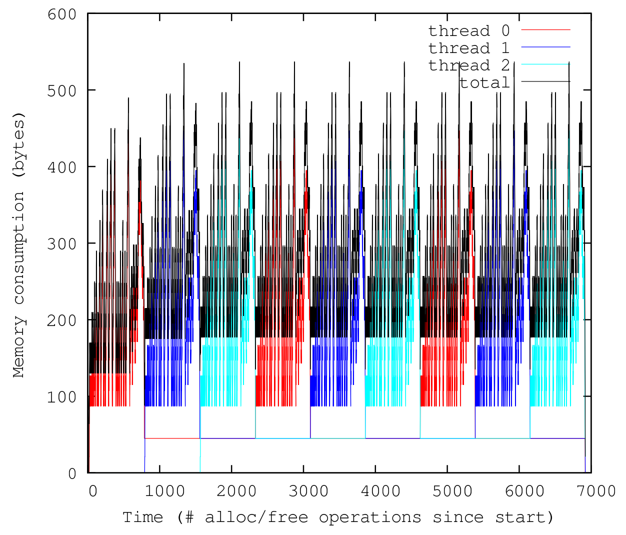

Figure 10. Garbage collection, synchronised.

Using a linked stack architecture in conjunction with synchronisation allows programs that would otherwise use large amounts of stack space to run with tight memory bounds, while leaving full control with the application programmer.

5.5 Garbage Collection Pause Times

Darjeeling uses a stop-the-world garbage collection algorithm, introducing pause times that may delay other time critical tasks in the system. We ran two experiments to measure these pause times. The first experiment used the garbage collection test from the previous section with the garbage collection phase indicated by setting/clearing a digital output which was monitoring with a digital scope. We observed pause times as long as 9 ms, which we consider worst case as the 2 KB heap contains a large number of tiny (5 byte) tree-node objects and the deep call stack contains many small frames. In a second test, modelling the average case, we allocated random sized byte arrays (10-50 bytes) at random intervals mimicking typical application behaviour of receiving radio packets (i.e., allocating packet buffers) and invoking the associated handler methods (i.e., allocating stack frames). For this test we observed a maximum pause time of 2.6 ms. For most applications this is satisfactory since the time-critical radio communication is handled by the native OS as part of an interrupt handler. When such interruptions cannot be tolerated, one should consider using an incremental garbage collector.

5.6 Portability

The portability of Darjeeling was evaluated by running our CTP implementation on three different test beds. Table 11 shows the different platforms and their properties. We have chosen to run Darjeeling on three different operating systems: TinyOS [15], Contiki [8] and FOS [6], each implementing a different form of concurrency.

We implemented native functions for the `javax.radio.Radio` class which has broadcast, unicast send, and receive primitives. Table 12 shows how many source lines of C code each of the ports required. The main component is the entry point of the application that bootstraps the virtual machine. The radio and sensors components are the native code required to access the MAC layer and sensor nodes from Java respectively.

Table 11. Platforms.

	TNOde	Tmote Sky	Fleck3(b)
MCU	ATmega128	MSP430	ATmega128
Architecture	8-bit	16-bit	8-bit
RAM	4 KB	10 KB	4/8 KB
Radio	CC1000	CC2420	nRF905
OS	TinyOS	Contiki	FOS
Concurrency	events	protothreads	threads

Table 12. SLOCCount for native code modules.

	FOS	TinyOS	Contiki
Main	225	359	86
Radio	69	92	69
Sensors	184	157	110
Total	478	608	265

The TNOde and Tmote Sky testbeds are indoor with a great amount of overlap, so that each node can hear a large number of neighbouring nodes. On each of the platforms we allocated the same amount of RAM (2 KB) to the virtual machine leaving the rest for the operating system. The Java application was programmed into the nodes via wired means.

6 A Real-World Application

In this section we evaluate the performance of Darjeeling for a real-world environmental monitoring application. The node, shown in Figure 11, is part of a small network that monitors micro-climate variation in an area of forest regeneration. The variables measured include soil moisture and leaf wetness, wind speed and direction, temperature and humidity, as well as engineering data regarding solar power generation, battery voltage and link quality. This application is typical of the majority of real-world sensors networks; the node is asleep most of the time, waking periodically (on the scale of minutes) to make measurements, perform some computation, then return results via a collection tree protocol.

In this section we describe the implementation of a collection tree protocol written entirely in Java, and also the re-implementation in Java of the sensor network application, originally written in C.

6.1 The CTP Implementation

Demonstrations exist where Java is used to simply glue C components together, but this does not show the true advantage of Java compared to an ASVM. To see how well Darjeeling copes with more complex multithreading applications we implemented a pure Java version of the well known Collection Tree Protocol (CTP), the standard routing protocol for TinyOS 2.x [10]. This is also a powerful demonstration of how higher-level languages like Java can be used to quickly and conveniently prototype algorithms. The implementation uses many advanced Java features including inheritance, multiple threads, synchronisation, generics, exception handling, and dynamic memory management.

The CTP library comprises three main components, the packet handler, routing engine, and data engine. The packet

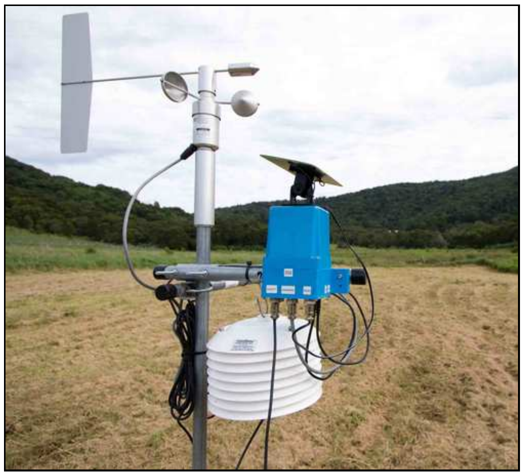

Figure 11. The environmental monitoring node which measures soil moisture, leaf wetness, wind speed and direction, temperature and humidity.

handler provides a packet sending and receiving interface to the other two components and supports reliable unicast with retries. It queues outgoing packets to maintain packet order. A queue of only a single incoming packet is kept on the native side of the implementation. While this could potentially cause incoming packets to be dropped if the virtual machine is not fast enough to process them, in practice we have not observed this problem. The packet handler provides an event interface that notifies listeners on successful or unsuccessful packet delivery, and packet reception.

Packets are stored as byte arrays wrapped in an instance of the `Packet` class, which has get- and set operations that directly mutate the data. We use inheritance to construct different packet types such as `CtpRoutingFrame` and `CtpDataFrame`. We use the factory design pattern to allow pluggable decoding of incoming packets

The routing engine uses adaptive beaconing and supports duplicate suppression and loop detection and repair. Features currently missing from our CTP implementation are congestion control and hysteresis in path selection. Link estimation is based on the average number of retransmits of unicast packets to a neighbour node. The data engine is responsible for forwarding and generating new data packets.

Our Java implementation uses five threads, two in the routing engine, two in the packet handler, and one in the data engine. The size of the code, as counted with SLOCCount, is 1,163 source lines of Java, divided over 20 source files. In comparison, the NesC implementation uses 1,613 source lines of code. The infused CTP library as a .di file is less than 8 KB (cf. Table 7).

6.2 Environmental Monitoring Application

The application was originally coded for the threaded operating system environment, FOS, with a CTP-like routing module implemented in C. The original C-code application contains 2 application threads and 6 system threads for CTP. The details are shown in Table 13. FOS stacks are fixed size and conservatively sized, but total RAM usage is well within the resources of an ATmega128 processor.

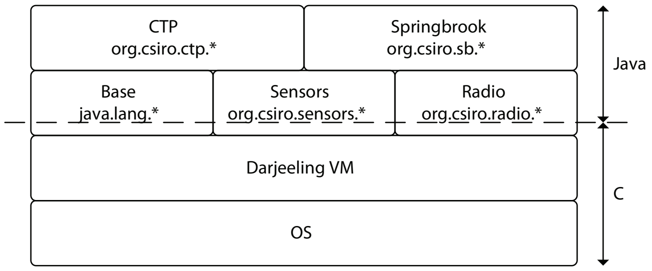

Figure 12. Software stack.

The application has been reimplemented in Darjeeling using the Java-based CTP. The software stack is shown in Figure 12. Characteristics of this implementation are also listed in Table 13. Stack space is still required for the underlying FOS system threads, one thread for the interpreter and one for communication with the radio. This leaves well over 2 KB of RAM available for the Java heap. The number of FOS threads has been reduced because the threads required for CTP are now Java threads whose stack space is allocated from the Java heap.

The Java version has less than half the source lines of the C version, mainly due to far fewer module initialisations and `#include` lines required, and is a single Java thread implementing a wait, measure, send cycle. The infused application as a .di file is 1,564 bytes, which is clearly small enough to conveniently load over the air. The run-time, CTP and application infusion combined easily fit within the resources of an ATmega128 processor. The flash memory for the Darjeeling case includes FOS and the infusion.

Each version of the application was run on the same node and the current consumption measured using a digital scope and a current shunt resistor. Data was downloaded from the scope for analysis, which allows for some precision in execution time measurement.

The measurement period is 5 minutes and the measurement cycle lasts around 2 seconds which is mostly delays for sensor settling time and wind-speed pulse accumulation. During the delays the processor is put into sleep state by FOS, for both the native C and Darjeeling versions of the application. In this regard the energy consumption is unchanged by using Darjeeling. The total ontime for the cycle is shown in Table 13, which indicates that Darjeeling is 50% slower than C, but over the measurement cycle of 5 minutes, this represents just 54 ms extra computation every 300s, which is an additional 0.018% overhead. The overhead in terms of energy consumption is even less as the CPU consumes only a fraction of the that consumed by the radio.

Table 13. Comparison of Springbrook application, 2 implementations.

	C	Darjeeling	units
Flash	43,3852	75,412	bytes
RAM (data+bss)	1,219	1,090	bytes
Stack	1,600	300	bytes
On-time	114	168	ms
SLOC	235	80	lines

Although in Section 5.1 we reported overhead as high as 100x for Java over C, the actual increase in processor wake-time is not nearly as bad. The time required for the radio to receive or transmit a packet is fixed, with the CPU essentially idling but awake. The time spent by the CPU doing actual processing is, in our application at least, relatively small compared to packet transmission times. Therefore the total wake-time is increased by much less than one would expect.

7 Conclusions

We have presented Darjeeling, a feature-rich virtual machine for wireless sensor networks which enables meaningful Java applications to run on sensor node hardware with memory sizes of 2-10 KB.

Darjeeling makes use of static linking, reducing executable sizes by 55-83%, but sacrificing support for reflection and the `synchronized` keyword on static methods. In the interest of memory efficiency we designed Darjeeling around a 16-bit architecture and dropped support for the `long`, `float`, and `double` types, although these could be added in the future. The infuser performs Java byte code analysis and maps the code onto a modified instruction set that includes support for direct manipulation of 16-bit values. Using this technique we were able to reduce stack space requirements by as much as 27% for a real-world environmental monitoring application.

Through strict separation of reference and non-reference types we are able to achieve precise garbage collection and compaction without the need for run-time type inspection using stack maps or type tagging. Our linked stack architecture enables threads running concurrently without the need for fixed, pre-allocated stack spaces, with threads using as little as 45 bytes when idle. Synchronisation between threads provides fine-grained control over stack space consumption.

The Darjeeling VM implementation is mostly platform agnostic and was easy to port to three different sensor node testbeds with different hardware and operating systems. Darjeeling can be used for serious applications as demonstrated by the pure Java re-implementation of an existing environmental monitoring application including a multithreaded many-to-one routing protocol. The results show that it is possible to use a virtual machine to prototype, develop, and deploy sensor networks applications.

In the future we would like to alleviate some of the current limitations of the Darjeeling VM. In particular, we would like to enhance the byte code analysis part of the infuser tool so we can support the `long` data type to improve compatibility with the full CLDC 1.0 specification. The current interpreter is not optimised for speed and we feel there is room for improvement in the performance department, which might have a positive effect on energy consumption as well. Finally we feel a generational garbage collector might help reduce collection pause times.

8 Acknowledgements

This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission (contract number FP7-2007-2-224053). The authors wish to thank members of the Springbrook project team: Tim Wark, Darren Moore and Ben Mackey. Furthermore we thank Guillaume Salagnac for his contributions to the VM, and Martin Bor for his support on the Contiki and TinyOS ports. Finally we like to thank our group members, our shepherd David Gay and the anonymous reviewers for their constructive comments on draft versions of this paper.

9 References

[1] Connected, limited device configuration specification version 1.0a. May 2000.

[2] Virtual machine specification, Java Card platform, v2.2.2, 2006.

[3] O. Agesen, D. Detlefs, J. Eliot, and B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl*, pages 269–279, 1998.

[4] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Introducing TakaTuka: A Java virtual machine for motes. In *SenSys '08: 6th ACM conf. on Embedded network sensor systems*, pages 399–400, New York, NY, 2008.

[5] A. Butters. Total cost of ownership: A comparison of C/C++ and Java, 2007.

[6] P. Corke. FOS: A new operating system for sensor networks. In *5th European Conference on Wireless Sensor Networks (EWSN08) (Poster)*, 2008.

[7] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *4th int. conf. on Embedded networked sensor systems*, pages 15–28. ACM New York, NY, 2006.

[8] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, FL, Nov. 2004.

[9] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo. The collection tree protocol (CTP). *TEP 123 Draft*, 2006.

[10] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *SenSys '09: 7th int. conf. on Embedded networked sensor systems*, Berkeley, CA, 2009.

[11] J. Koshy and R. Pandey. VMSTAR: Synthesizing scalable runtime environments for sensor networks. In *SenSys '05: 3rd int. conf. on Embedded networked sensor systems*, pages 243–254, New York, NY, 2005.

[12] E. Lafortune. http://proguard.sourceforge.net/.

[13] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, 2002.

[14] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI'05: 2nd Symposium on Networked Systems Design & Implementation*, pages 343–356, Berkeley, CA, 2005.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, et al. TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[16] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, Apr. 1999.

[17] T. Lindholm and F. Yellin. Java Card™ 2.2 off-card verifier, 2005.

[18] D. Palmer, P. Sikka, P. Valencia, and P. Corke. An optimising compiler for generated tiny virtual machines. In *2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, pages 161–162, May 2005.

[19] Sentilla. http://www.sentilla.com.

[20] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: The squawk Java virtual machine. In *VEE '06: 2nd int. conf. on Virtual Execution Environments*, pages 78–88, New York, NY, 2006.

[21] J. Solorzano. leJOS: Java based OS for Lego RCX.

[22] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP '03: 19th ACM Symposium on Operating Systems Principles*, pages 268–281, New York, NY, 2003.