A Virtual Machine for the Insense Language

Callum Cameron, Paul Harvey, and Joseph Sventek
cjcameron7@gmail.com, p.harvey.1@research.gla.ac.uk, joseph.sventek@glasgow.ac.uk

School of Computing Science, University of Glasgow, Scotland

***Abstract*—The Insense VM is a specialised Java virtual machine for running Insense programs on wireless sensor nodes. The VM runs on top of InceOS, a purpose-built operating system. A split VM architecture is used, in which Insense programs are compiled to Java classes, then linked and compacted on a more powerful machine into a form suitable for execution by the VM. Measurements demonstrate that the virtual machine achieves good performance and memory usage for realistic Insense programs.**

I. Introduction

The Insense language [3] has been developed to ease the programming of wireless sensor networks. Insense uses a component-based model of concurrency, where components are strictly encapsulated and communicate through message passing. Insense is supported by a purpose-built operating system, InceOS [11], which runs on the Tmote Sky/TelosB platform. This paper presents the Insense VM, a specialised Java virtual machine which runs on top of InceOS and directly supports the features of Insense. Insense programs are compiled to Java bytecode, then linked and compacted into a much smaller form prior to installation on a sensor node.

Sensor nodes have extremely limited resources; each node typically has a few kilobytes of RAM, and a processor running at several megahertz. Using a virtual machine introduces a performance and memory overhead over native execution. In a highly constrained environment, it is essential that these overheads do not compromise the usefulness of the system. We demonstrate that the Insense VM does not suffer from this problem, and is capable of running realistic and useful Insense programs.

II. Related Work

A. Insense

Widely-used operating systems for wireless sensor networks impose unusual programming models to compensate for the limited resources available. For example, TinyOS [12] uses the nesC language, with an event-driven 'split-phase' programming model. In nesC, all operations are non-blocking, and programs use many callbacks which can make the flow of control difficult to follow. Contiki [7] programs are written in C, but use macros and continuations to simulate a traditional threaded environment on top of an event-driven core. Programmers using these systems must have extensive knowledge of low-level and embedded programming. Domain experts wishing to use wireless sensor networks in their own fields – often described as the intended users of these systems – are unlikely to have this knowledge.

The Insense language is designed to overcome this problem [3], [9], [13]. Insense is a component-based language. A program consists of several *components*. A component is essentially a thread which executes a *behaviour* function repeatedly, and has private state. Components can only communicate by passing *messages* over strongly-typed *channels*, which use a blocking 'rendezvous' model. Senders and receivers block on a channel until both are present, at which point the message is passed and both components are unblocked. Channels prevent many common synchronisation problems encountered in shared-memory concurrent programming. Execution begins in a *primordial main* function, which creates and connects the initial set of components. Insense uses garbage collection to reclaim unused memory. The Insense standard library defines functions and components for accessing sensors, the radio, and other hardware present on a sensor node. These components use channels in the same way as user-defined components.

Fig. 1 shows an example Insense program. The `Sender` component sends integers over a channel to the `Receiver` component. Channels consist of two *half channels* connected together. *In* channels are connected to *out* channels of the same type.

B. InceOS

The first Insense implementation compiled programs into C, targeting Contiki [9]. However, the conflicting designs of Contiki and Insense led to high overheads [10]. To overcome this, the InceOS operating system was developed, which supports the semantics of Insense programs directly [11].

InceOS is a preemptive multitasking operating system where the unit of execution is the Insense component. Communication over channels is supported through system calls. The blocking semantics of channels are used to implement a lightweight round-robin scheduler. Garbage collection is provided through a reference counting system. The OS is implemented in C, and includes the Insense standard library functions and components. Currently, InceOS runs on the Tmote Sky/TelosB platform, with a Texas Instruments MSP430 microcontroller, and in the Cooja simulator provided by Contiki.

The second Insense implementation compiled programs into C, targeting InceOS. The C code and the OS are linked into a single monolithic binary image. However, this approach is inflexible and does not support dynamic reprogramming. This motivated the development of the Insense VM.

C. Virtual Machines for Constrained Devices

Virtual machines offer improved flexibility and robustness over native execution. Targeting a VM means that software can

```
type ISender is interface(out integer output)
type IReceiver is interface(in integer input)

component Sender presents ISender {
    number = 0;

    constructor() {}

    behaviour {
        send number on output;
        number := number + 1;
    }
}

component Receiver presents IReceiver {
    constructor() {}

    behaviour {
        receive number from input;
        printString("Got value ");
        printInt(number);
        printString("\n");
    }
}

// Primordial main
s = new Sender();
r = new Receiver();
connect r.input to s.output;
```

Fig. 1. An example Insense program.

run on a variety of machines without recompilation (assuming a VM is available on each platform). A VM specification can be implemented by different execution methods, such as interpretation and JIT compilation, with characteristics suited to different situations. VMs improve robustness by preventing undefined behaviour, and detecting runtime errors which would go undetected in native code. This is particularly important for sensor nodes, which have no hardware memory protection. All code runs in a single address space and privilege level, so a misbehaving native program can corrupt the entire system. A VM can detect and handle such errors safely.

Specific to sensor nodes, VMs can ease *over-the-air dynamic reprogramming*, where new code is sent over the radio and installed on a node after deployment. Dynamic reprogramming has been done with native programs [7], but a VM has the advantage of denser code, which is less expensive to send over the radio. It is also easier to link new code into a program running in a VM than it is to update native code at runtime.

Several VMs have been developed for use on wireless sensor nodes. Many have used the Java bytecode instruction set, which is well supported by existing tools and libraries, and is well understood. Examining these VMs reveals certain trends and characteristics.

The full Java virtual machine (JVM) specification [15] is too complex to support on embedded devices. The *Java Platform, Micro Edition* (Java ME) is designed for smaller devices than the full specification, and offers a complete VM with a reduced standard library. However, even these implementations are too large – on the order of hundreds of kilobytes – for use on sensor nodes. JVMs designed for sensor nodes deliberately exclude parts of the specification in order to reduce size, e.g. threads, reflection [6], floating-point arithmetic, exceptions, garbage collection [14], and user-defined classes [8]. Optimisation is for space rather than speed [4]. All use a much reduced standard library.

Java class files are usually large, and only a small fraction of this is bytecode (see Section IV). The rest is metadata used for dynamic linking. Because memory on sensor nodes is limited, and data transmission is costly, many VMs use a *split VM architecture* [18], [4], [2], where class files are linked on a desktop machine into a much smaller and denser format. The VM running on the node executes this format rather than the original class files.

Of note is Darjeeling [4], an open source VM for the MSP430 and other architectures that is similar to the Insense VM. However, the goals of Darjeeling are different; users are expected to program in Java. The standard thread-based Java concurrency mechanisms are supported, as is a small subset of the Java standard library. Darjeeling runs on top of TinyOS and Contiki, among others.

The Insense VM, in contrast, is intended to support Insense and to integrate with InceOS. The Insense concurrency model is supported instead of threads and synchronisation, and InceOS provides its own standard library.

III. The Insense VM

The Insense VM is a specialised Java virtual machine designed to run Insense programs on MSP430-based sensor nodes, on top of InceOS. It is a 16-bit, interpreted, stack-based VM. It uses the 'split VM' approach to run Java bytecode programs on devices with as little as 10 kB of RAM and tens of kilobytes of program memory. This section discusses the key features of the VM.

A. JVM Feature Set

The Insense VM does not implement the whole JVM specification [15]. Many of the features of a standard JVM are unnecessary for Insense, and would increase the memory footprint of the VM and interpreted programs by an unacceptable amount. Features not implemented include:

- *Interfaces* – Insense interfaces add fields (channels), not methods, to a component. Insense is not object-oriented, so does not need interfaces in the Java sense. The VM does support inheritance of classes, which is used internally but not exposed in the language.
- *Synchronisation* – the Insense concurrency model replaces traditional Java synchronisation.
- *Data types* – several primitive data types are unsupported (see Section III-C).
- *The Java standard library* – Insense defines its own standard library. Only the minimum of classes from the `java.lang` package are supported (`Object`, `String`, and the primitive wrapper classes), with as few methods as possible.
- *Reflection* – neither the APIs nor the runtime type information for reflection are supported. The only type information associated with an object at runtime is its class, which is used by the garbage collector.

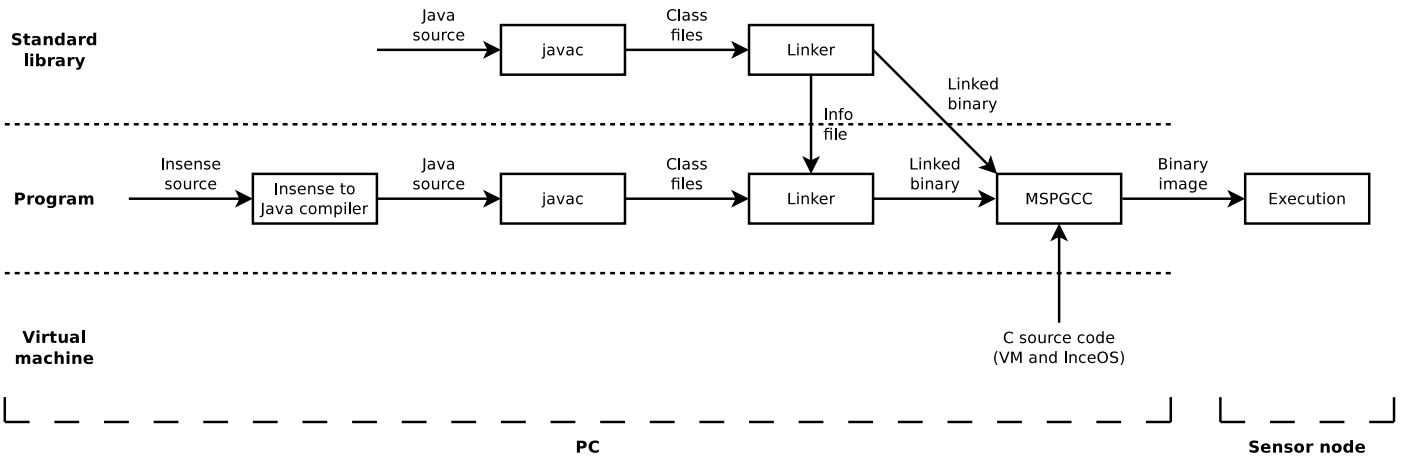

Fig. 2. Compiling and executing an Insense program. The 'split VM' architecture is shown by the separation between PC and sensor node.

Bytecode instructions used only by these features are not implemented. Some features, such as multidimensional arrays and exceptions, are supported in a restricted form.

B. Compilation and Linking

The Insense VM uses a split VM model, similar to the VMs discussed in Section II.

A traditional JVM uses 'lazy loading' of class files. Classes are compiled independently, and all references to other classes are symbolic. The JVM loads a class file when it is first referenced, and resolves all symbolic references before continuing execution. However, this is a demanding process, and class files are often larger than the whole main memory of a sensor node.

The essence of the split VM approach is to resolve all references offline, on a more powerful machine, and link the class files into a single file which the VM can execute. This places less demand on the sensor node. The linked file can be smaller than the original class files by an order of magnitude or more. The techniques used in the linker to achieve these reductions are discussed in Section IV.

The split VM system used by the Insense VM has five steps, shown in the 'program' section of Fig. 2.

1) Insense code is compiled to Java source code. Each component is a Java class.
2) The Java source is compiled to Java bytecode using the `javac` compiler.
3) The linker is run twice; first to link the standard library, and second to link the bytecode program against the standard library.
4) The VM is compiled using MSPGCC. The output of the linker is statically compiled into the C program at this stage, and stored in program memory[1].
5) The resulting binary is installed on a mote or run in simulation.

[1]Dynamic over-the-air reprogramming has not been implemented yet, and is discussed as future work in Section VI. With support for dynamic reprogramming, only the standard library would be compiled into the VM.

The linker produces a second output: a symbolic information file which allows other programs to link against the corresponding binary. This is used to separate the standard library from programs. The standard library contains classes which programs rely on, such as `Object` and `Integer`, native method declarations, and wrapper classes for native components.

C. Memory Model

The Insense VM is stack-based, as in a standard JVM. An alternative register-based model was rejected because of memory limitations. Stack-based bytecode tends to be slower but smaller than equivalent register-based code [17], and the scarce memory on sensor nodes encourages optimisation for space rather than speed.

1) Slot Size: To support the stack-based model, every call stack frame has an operand stack consisting of fixed-size *slots*. Local variables are stored in separate slots of the same size. The JVM specification requires slots to be 32 bits, to match the word size of common desktop CPUs. Values of all data types occupy one slot, except for `long` and `double`, which occupy two.

However, the specification defines instructions in terms of the number of slots they operate on, without reference to the size of the slots. This means the slot size can be changed without modifying the bytecode, as long as each data type still occupies the same number of slots as before.

The Insense VM uses a 16-bit slot size, to match the native word size of the MSP430 and the largest integer type in Insense. This allows a convenient mapping between Insense, Java, and MSPGCC data types, as shown in Table I, while saving memory over the standard 32-bit slots. The integral types all use one slot. MSPGCC's 32-bit `float` represents Java's `double`, which occupies two slots. The VM does not support Java's `long`, because Insense does not have a 32-bit integer type. As in a standard JVM, `byte` values occupy a whole slot.

Changing the slot size changes the range of values a type can have. However, the VM is intended only to run Insense

TABLE I. MAPPING FROM INSENSE TO JAVA TO MSPGCC DATA TYPES.

Insense type	Java type	Size (slots)	MSPGCC type	Size (bits)
integer, unsigned, boolean	int	1	int	16
byte	byte	1	char	8
real	double	2	float	32
reference	reference	1	void*	16

programs, not existing Java programs which might depend on the larger range available in a standard JVM.

2) Stack Frames: A new call stack frame is allocated on the heap when a method is called, rather than being pre-allocated at component creation. A minimal stack frame with no slots and no local variables occupies 16 bytes of RAM. Each additional slot or variable requires two bytes. The number of slots and variables used by a method is known at link time, so the whole frame can be allocated as a single unit. Memory must be allocated in advance for a component's C stack, which is used to run the interpreter and native methods. This is typically on the order of several hundred bytes per component.

3) Objects: All objects are allocated on the heap. A class definition includes a reference to the class's superclass (as in standard Java, all classes descend ultimately from `Object`), the size of its fields, and a virtual method table.

When an object is instantiated, space is allocated for its fields. Unlike stack slots, fields can differ in size, and are packed in memory. The size of a field must therefore be known when accessing it. A standard JVM keeps this information in the constant pool, but the split VM approach means that the Insense VM does not have a constant pool. Instead, new type-specific versions of the `getfield` and `putfield` instructions have been introduced for the different field sizes. The instruction to use in each case is chosen at link time.

Some classes are treated specially. `String` contains a pointer to a native string. Arrays contain a pointer to a native array, as well as the size, dimensions, and element class of the array. The class of an array itself is the special placeholder 'array' class, and variants of the `instanceof` and `checkcast` instructions have been introduced to test the element type and dimensionality of arrays. Additional space is allocated for these classes by the VM.

4) Static Fields: Insense does not support static fields because they could break the strict encapsulation of components. Hence, they are not allowed in bytecode programs. However, they are useful in the standard library as global references to system components. Because the standard library is designed to be compiled into the VM image, space for static fields is allocated at compile time. As with instance fields, static fields are packed in memory, and new versions of the `getstatic` and `putstatic` instructions are used to access fields of different sizes.

5) Garbage Collection: The VM uses the reference counting garbage collector provided by InceOS. All Java objects are reference counted, and bytecode instructions which manipulate objects increment and decrement the reference counts appropriately. A bitmap in each stack frame keeps track of which slots and variables contain objects, so that their reference counts can be decremented when a method returns. Fields in classes are handled in a similar way, except that this information is known at link time, and is stored directly in the linked binary.

Like any simple reference counting system, the InceOS collector cannot collect cycles in the object graph. However, Insense prevents the creation of cycles by enforcing certain rules. Insense structures cannot reference other structures, and all complex data types are duplicated when sent over channels.

Potential enhancements to the garbage collector are proposed as future work in Section VI.

D. Support for Insense Features

Previous sections have examined how the VM supports the Java execution model. This section discusses features specific to Insense.

1) Components: Each component in an Insense program is compiled into a Java class extending the standard `Component` class. Channels are stored as fields. A component's constructors run in the context of the creating component, and call the native `start` method to begin execution of the new component as an independent entity.

Each Java component is run by an instance of the native 'interpreter' component, which is written in C and is the central part of the VM. The Java primordial main is interpreted from InceOS's own primordial main, which also performs system initialisation. As components are the basic schedulable entities in InceOS, the scheduler ensures that the execution of interpreted components is interleaved with each other and with system components.

2) Channels: The standard library provides generic `ChannelIn` and `ChannelOut` classes which are instantiated for each channel in a component. Using generics means that the `javac` compiler type-checks all channel operations at compile time, so the VM does not have to do any type-checking at runtime.

Channels are one of the classes treated specially by the allocator. A Java channel wraps a native channel, and interpreted programs use native methods to access the underlying functionality. Data is passed opaquely from sender to receiver, so the size of the type being passed must be specified when a channel is created. An advantage of generics in this respect is that all Java channels simply pass a pointer, so the VM can create channels without any type information being available at runtime. A drawback to using generics is that primitives must be 'boxed' and 'unboxed' before and after being passed over a channel, which can be time-consuming if a channel is used frequently.

TABLE II. SIZES OF INSENSE PROGRAMS AS CLASS FILES AND LINKED BINARIES.

Program	Class files (B)	Linked binary (B)	Linked binary (% of class files)
Standard library (basic)	19500	436	2.2%
Standard library (full)	19500	770	3.9%
Arithmetic	2032	172	8.5%
Blink	2977	321	10.8%
Integer channel	3376	273	8.1%
Radio sense to LEDs	4515	374	8.3%
3D array channel	3961	533	13.5%
Sense	3238	292	9.0%
Data Logger	16272	2873	17.7%
Base Station	8796	2030	23.1%

E. Interaction with Native Code

Most of the Insense standard library is implemented in C and accessed through Java native methods. Java components which wrap system components also need special treatment.

1) Native Methods: Methods in the standard library can be declared as native. In a standard JVM, native methods are called using the same instructions as normal methods, and the constant pool is used to resolve and load the necessary native code. Because the Insense VM does not have a constant pool, a new `invokenative` instruction has been introduced specifically for calling native methods. This instruction behaves like `invokestatic`, except that its argument is treated as identifying a native method rather than an interpreted method. The structure of most native methods is to extract native data from the arguments, to make a system call, and to convert the result back into Java form. The existing invocation instructions `invokestatic`, `invokespecial`, and `invokevirtual` are only used to invoke interpreted methods.

This approach means that native methods must be static. There is currently no support for virtual invocation of native methods, but this has not been found necessary in implementing Insense programs.

2) Native Components: The Insense standard library defines components for accessing the sensors, timers, radio, and other hardware. InceOS implements these components in C. Interpreted code accesses these components through the channel mechanism, as normal, using Java channels which wrap the components' native channels. Because Java channels expect to send and receive references to Java objects, but native channels use a variety of C data types, conversion between types is performed within the channel mechanism for any transfer between the two kinds of channels.

IV. BYTECODE COMPACTION

This section examines the techniques used in the linker to reduce the size of class files. These are examples of *compaction*, rather than the *compression* used in standard JAR files. Decompression is an expensive task to perform on a sensor node, whereas compaction simply discards unnecessary parts of the data, leaving the rest immediately available for use.

Table II presents the sizes of the class files and linked binaries for the standard library and various programs, generated using the techniques discussed below. Typically, linked binaries for programs are around a tenth to a quarter of the size of the corresponding class files.

TABLE III. AVERAGE PERCENTAGE COMPOSITION OF CLASS FILES. 'USED' AND 'UNUSED' INDICATE WHETHER INFORMATION IS PRESENT IN THE LINKED BINARY.

Data	Standard library (26 class files)	Programs (37 class files)
Constant pool	68.1%	64.0%
Class metadata used	0.3%	0.2%
Class metadata unused	5.9%	4.1%
Field metadata used	0.3%	0.2%
Field metadata unused	10.9%	2.5%
Method metadata used	1.9%	2.1%
Bytecode	2.4%	13.0%
Method metadata unused	10.3%	13.9%
Method total	14.6%	29.1%
Total used	4.8%	15.4%
Total unused	95.2%	84.6%

A. Composition of Class Files

Java class files contain member definitions (fields and methods), metadata, and constant pools. Past analyses of Java programs show that, on average, class files can be as little as 33% method definitions [1], and only 20% bytecode [16]. However, this may not be representative of Insense programs, and in particular the standard library, which contains mainly class and native method definitions.

The class files from the standard library and various Insense programs have been analysed to find how much of their data can be discarded. The results are shown in Table III. Most of the unused data is related to linking, and is unnecessary after being used by the linker. The rest is mainly metadata related to Java features unsupported by the VM, or which is encoded in the VM's new instructions (e.g. the size of fields, and whether methods are native). Much of the unused data is retained in the program's symbolic information file, which is substantially larger than the linked binary, but is only used by the linker and never installed on a sensor node.

B. Optimisations

The linker performs various optimisations on the data which is kept, to further reduce the linked binary's size.

1) 'Static Only' Classes: In Java, all methods and fields must be part of a class; there is no such thing as a 'top-level method' or a global variable. However, some classes contain only static methods and static fields, are never instantiated, never extended, and never used as the type of a variable. Because all symbolic references are resolved in the linker, the

TABLE IV. SIZE OF THE DEFAULT VM CONFIGURATION. RAM USAGE REFERS TO THE STATICALLY ALLOCATED 'DATA' AND 'BSS' SECTIONS.

Interpreter (bytes in ROM)	Interpreter (bytes in RAM)	Standard library (bytes in ROM)
27980	471	436

linked binary does not need to include a class definition for these classes at all; only the methods and fields themselves are included. A class suitable for this treatment is marked with the custom 'static only' annotation, which is detected by the linker. All wrapper classes for native components are marked as static only, giving significant space savings.

2) Empty Methods: Java object construction takes place in two stages. The `new` instruction allocates the memory required for the object, and then the constructor is called. All Java classes are required to have a constructor. The `javac` compiler generates default constructors where necessary, which do nothing but call the superclass's constructor, to ensure this rule is satisfied. This leads to long chains of calls to methods that do no useful work at all.

The linker detects these methods and removes them from the linked binary. All calls to these methods are also removed. This continues iteratively until all constructors and static methods which do nothing, transitively, have been removed. Some modification of the bytecode around the removed call sites is necessary to ensure correct execution. In particular, arguments to the method are pushed to the operand stack before a call, and must be disposed of now that the call no longer happens. If the argument was pushed immediately before the call, the pushing instruction is removed; otherwise, an appropriate number of `pop` instructions are inserted.

Virtual methods are not removed even if they do nothing, so that virtual calls continue to work as expected.

3) Native Methods: The `invokenative` instruction refers to native methods by IDs assigned by the linker, made available to the VM through a C header file. Hence no information about native methods is stored in the linked binary at all.

C. Conditional Compilation

InceOS provides many standard library functions and components, only some of which might be used by a given program. The original Insense to C compiler (see Section II) relied on the C linker to remove unreferenced functions. The VM, however, must reference everything, which uses most of the Tmote Sky's 48 kB program memory and leaves little space for bytecode.

This is overcome using conditional compilation in the linker and the VM. Only the core of the system is enabled by default, and the user enables any additional components required. Because the standard Java toolchain does not support conditional compilation, a custom 'ifdef' annotation has been introduced which is used by the linker to mimic the behaviour of the C preprocessor directive. The VM build system ensures that a matching set of native components is compiled. Any interpreted program which only uses these components can then be run without having to modify the VM image or the standard library installed on a mote.

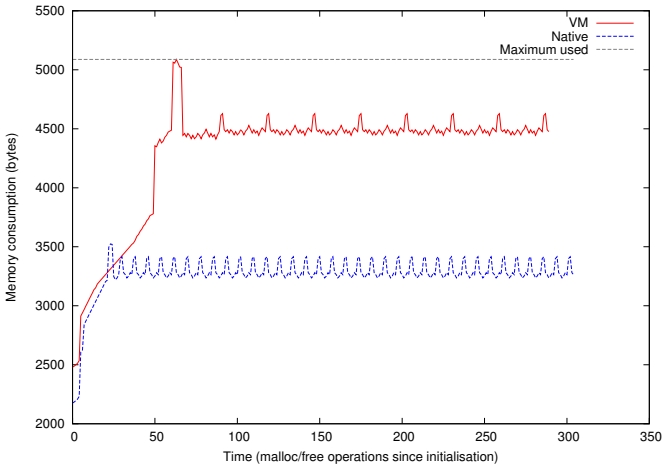

Fig. 3. Dynamic memory usage of the *RadioSenseToLeds* application.

Table IV shows the static memory consumption of the interpreter and standard library in the default configuration. Table V shows the additional memory required to support each of the optional components. (Dynamic memory usage of complete programs is examined in Section V.)

V. EVALUATION

A. Memory Usage

The static memory usage of Insense programs has been examined in Section IV. Insense programs also allocate memory dynamically from the heap. Because the garbage collector uses reference counting, memory is freed as soon as it is no longer referenced.

1) RadioSenseToLeds: In Fig. 3, the 'VM' plot shows the dynamic memory usage of the *RadioSenseToLeds* application over time, running on the Insense VM. This program takes readings from one of the on-board sensors and broadcasts them over the radio. A second component receives messages from other nodes and displays the bottom three bits of the reading on the LEDs. Time is measured in the number of allocations and deallocations since the end of the OS initialisation sequence.

Memory tracking starts at the beginning of the primordial main, after system initialisation. Most of the memory allocated before this is used for the C call stacks of the system components and the primordial main. The large increases at around times 50 and 60 are the creation of the sender and receiver components (again, most of the memory is used for the C stacks). The large drop at 70 is the termination of the primordial main. Periodic usage is observed after about 100, as the sender's behaviour is executed repeatedly.

The 'Native' plot shows the behaviour of the same program, compiled using the original Insense to C compiler. The memory usage is considerably lower, mainly because the stack size needed for each component is calculated and set in the C code. This is in contrast to the VM, where the stack size of each component must be large enough to run the interpreter, regardless of which Insense program is being run. However, it should be noted that there is still more than enough memory available to run the interpreted program.

TABLE V. ADDITIONAL MEMORY REQUIREMENTS FOR OPTIONAL COMPONENTS. RAM USAGE REFERS TO THE STATICALLY ALLOCATED 'DATA' AND 'BSS' SECTIONS.

Component	Interpreter (bytes in ROM)	Interpreter (bytes in RAM)	Standard library (bytes in ROM)
Basic sensors	1810	27	0
Additional sensor functions	362	0	0
Additional sensor component	1458	31	0
Acceleration X-axis	1309	13	47
Acceleration Y-axis	1301	11	47
Pressure	1307	13	47
Tilt X-axis	1301	11	47
Tilt Y-axis	1301	11	47
Button	298	15	0
Exceptions	305	0	45
Timer	605	4	69
Radio	3444	100	64
Runtime error checking	983	2	45
Serial comms	522	93	0
Storage	2739	31	109
Printf	2376	28	0
Fourier	1064	200	0
Diagnostics	1946	12	0

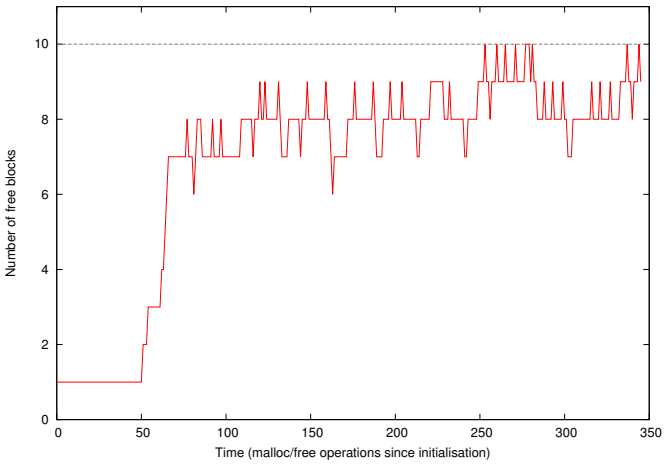

Fig. 4. External fragmentation: number of free blocks in the interpreted *RadioSenseToLeds* application.

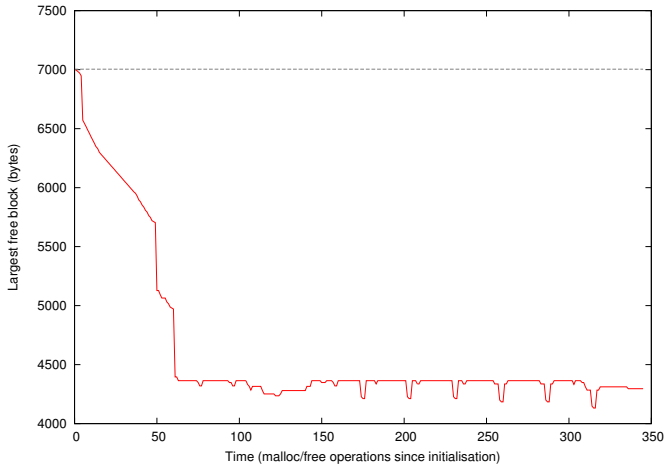

Fig. 5. External fragmentation: largest free block in the interpreted *RadioSenseToLeds* application.

As memory is allocated and freed, the region of memory used by the allocator becomes fragmented. Fig. 4 shows the total number of free blocks, and Fig. 5 shows the size of the largest free block, in the interpreted *RadioSenseToLeds* program over the same time period as Fig. 3. Although memory is being allocated and freed throughout this time, the level of fragmentation is bounded, and does not increase to the point where it becomes problematic.

Table VI compares the total static memory requirements for the interpreted and native versions of *RadioSenseToLeds*. For the interpreted version, ROM usage includes the linked standard library, the bytecode program, and all C code. Again, the native version is considerably smaller, but there is still enough memory available for the VM and interpreted application.

2) DataLogger: Fig. 6 shows the dynamic memory usage of a more complex interpreted program, *DataLogger*. This program reads data from the sensors and stores it in flash.

TABLE VI. STATICALLY ALLOCATED MEMORY FOR NATIVE AND INTERPRETED *RadioSenseToLeds*.

Type	RAM (bytes)	ROM (bytes)
Interpreted	632	38982
Native	590	22440

Later, it can send the stored data over the radio to a base station running on another node. A user controls the program by sending commands over a serial connection to the base station, which parses the commands and sends the appropriate radio messages to the data logger. This program uses many of the optional components provided by the standard library, including the acceleration and tilt sensors, the radio, the storage module, timers, and runtime error checking.

Between times 500 and 3500, the program is reading from the sensors. The large spikes during this time are buffers filled with data being passed to the flash component; these are arrays which are copied when they are sent over a channel, in

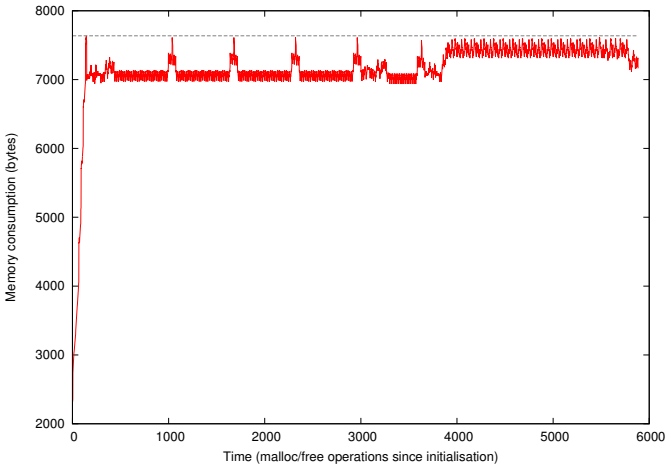

Fig. 6. Dynamic memory usage of the *DataLogger* application.

TABLE VII. STATICALLY ALLOCATED MEMORY FOR *DataLogger* AND THE BASE STATION.

Program	RAM (bytes)	ROM (bytes)
DataLogger	636	42866
Base Station	724	41566

keeping with the strict encapsulation of components. Between about 4000 and 6000, data is being streamed from flash to the radio. Table VII shows the static memory usage for *DataLogger* and the base station. Despite the complexity of the application, there is still more than 2 kB of RAM and 6 kB of program memory free. This demonstrates that the Insense VM is capable of running a realistic and potentially useful application.

B. Performance

1) Instructions per Second: One measure of the VM's performance is the number of bytecode instructions executed per second. The VM was instrumented to capture this data, and several programs were run. These measurements were taken on a Tmote Sky sensor node running at 4 MHz. Table VIII shows the results.

Different bytecode instructions take different times to execute. The number of instructions a program executes per second depends on which instructions it uses. *RadioSenseToLeds* is the same application used in Section V-A1. It is largely I/O bound, and spends most of the time communicating with the sensors and the radio. *Fourier* repeatedly performs a fast Fourier transform on an array of integers, and consists mainly of integer arithmetic and array accesses. *Arithmetic* simply performs integer arithmetic in a loop; this is perhaps unrealistic, but it demonstrates the VM's highest speed.

The table also shows the effect of runtime error checking on instruction throughput. With runtime error checking enabled, the VM detects various problems as they occur, and throws an appropriate exception. Checks include testing for null pointers, testing for out-of-bounds array accesses, and reporting out-of-memory conditions. If exceptions are not caught by the interpreted program, the offending component is terminated. With runtime checks disabled, execution continues after an error with undefined behaviour.

TABLE VIII. INSTRUCTIONS PER SECOND.

Program	Runtime checks	No runtime checks
RadioSenseToLeds	128	132
Fourier	29051	29176
Arithmetic	42894	42871

TABLE IX. RUNNING TIMES OF INTERPRETED AND NATIVE PROGRAMS.

Program	Time (s)	Standard Deviation (s)
RadioSenseToLeds (VM)	0.233	0.001
RadioSenseToLeds (Native)	0.230	0.001
Fourier (VM)	2.420	0.001
Fourier (Native)	0.161	0.000

As shown in the table, runtime error checking does not have a significant performance overhead. Thus the only reason to disable error checking is if the memory saved by doing so is needed for bytecode.

2) Execution Time: Counting instructions per second is an artificial measure of performance. A more important measure is the time taken for a program to do something useful; it is unimportant how many instructions are executed in the process. Table IX shows the running times for several programs, both as interpreted programs running on the VM, and as native programs. The figures are the average running times of one hundred executions of each program's behaviour. For each of the interpreted programs, runtime error checking is enabled.

For a computationally intensive task such as the fast Fourier transform, native code is more than an order of magnitude faster than interpreted code. In contrast, for an I/O bound program such as *RadioSenseToLeds*, the overhead of interpretation is negligible.

A sensor node VM would be expected to run mainly I/O bound applications. It is unlikely that computationally intensive tasks, such as the fast Fourier transform, would be written in interpreted code. In fact, the C implementation used by the native Fourier example is available to interpreted programs as a native method in the standard library. Similar computationally intensive tasks, where the overhead of interpretation is large, could be implemented in C and exposed to the high-level interpreted program in the same way. This demonstrates that the VM is fast enough for its intended use.

VI. FUTURE WORK

Although the Insense VM can execute Insense programs effectively, it could be improved in several ways.

A. Dynamic Reprogramming

As described in Section II, one of the advantages of VMs over native code is that dynamic over-the-air reprogramming can be supported more easily. Bytecode programs are usually smaller and less expensive to send over the radio than equivalent native code. Although dynamic reprogramming has not been implemented yet, the Insense VM has been designed with it in mind. The split between the standard library and user code means that programs are already in a format which

could be used for dynamic reprogramming, with appropriate support from the system.

Several changes are required to support dynamic reprogramming in the VM:

- Bytecode programs are stored in program memory, which is flash that can only be erased and rewritten in pages. The program would be located at a known page-aligned location (rather than contiguous with the end of the standard library, as at present), so that it could be erased and replaced.
- Additional native code would be required to send and receive bytecode programs over the radio, to buffer them until fully received, and to handle the replacement process.
- The semantics of unloading a program would have to be defined and implemented: when code is erased, what happens to instances of classes defined in that code?

Support for dynamic reprogramming would also be added to the language. This could either be through native methods, or through integration with the channel mechanism, so that dynamic reprogramming becomes a case of sending code over a channel to another node. Whichever option is chosen, the distinction must be made between *weak mobility*, where only the code is sent to another node, and *strong mobility*, where both code and state are sent – in effect, migrating a running component between nodes.

B. Compiler

As described in Section III, Insense programs are currently compiled to Java, and then to bytecode using `javac`. There would be several advantages in compiling Insense directly to bytecode:

- *Reduction in code size* – some of the classes and methods in the standard library exist only for compatibility with the code generated by `javac`. Certain methods in the 'primitive classes' such as `Integer`, and classes to support exceptions, are not strictly necessary, but are required to link with classes generated by `javac`.
- *Optimisation of bytecode* – the code generated by `javac` might not represent Insense idioms in the most efficient way. Optimising at the compilation stage, with knowledge of the changes to the instruction set made by the VM, might be easier than attempting to optimise `javac`-generated bytecode in the linker.
- *Variable and stack usage* – the Insense VM uses bitmaps to record which local variables and stack slots contain objects, so that garbage collection works correctly. These must be maintained at runtime because `javac`-generated code reuses slots for different types throughout the lifetime of a method call. In Darjeeling, in contrast, a frame has separate variables and stacks for objects and primitives, so that the information needed for garbage collection is available statically [5]. This is possible because of the extensive bytecode rewriting Darjeeling performs during the linking stage. A similar process could be adopted here.

C. Garbage Collection

The Insense VM currently uses a reference counting garbage collector, as described in Section III. As with all simple reference counting systems, it cannot collect cycles in the object graph. The Insense language prevents the occurrence of cycles. However, if the VM was to be used for hand-written Java or other languages, it might be necessary for the garbage collector to handle cycles correctly.

Either the reference counting system could be replaced by a tracing collector, or the current system could incorporate tracing as a 'backup' collector, used as a last attempt to free memory when an allocation fails. Alternatively, cycle detection could be added to the existing reference counter.

However, each of these requires additional memory. An advantage of the current system is that memory is reclaimed as soon as possible, leaving space for other components to use. Since memory is such a scarce resource, this is desirable, and avoids the long pauses needed for a tracing collector. Thus the best way to add cycle handling might be as a conditionally compiled option, with the current system as the default.

D. Linker File Format

The linked binary file format was designed to be as small as possible, but this comes at the expense of flexibility. The current format does not support relocatable code, as all references are absolute. In a dynamic reprogramming system, when multiple bytecode programs are in use, either each must have a particular space in memory reserved for it (which could be wasteful), or the programs must be mutually exclusive. Reprogramming at the granularity of individual classes would be difficult, if not impossible.

To overcome this, ideas could be adopted from the Darjeeling equivalent (referred to as an *infusion*) [4]. Rather than splitting programs into a standard library and user code, Darjeeling allows composition from an arbitrary number of modules (one of which is the standard library). References between modules are resolved using a symbol table, where references identify a module and symbol number within that module, rather than being absolute offsets. This allows modules to be relocatable.

Relocatable modules require additional memory to store the various tables, and additional information in the file format to allow for the tables to be constructed. Any modification to the Insense VM's file format would therefore be a tradeoff between the flexibility of a Darjeeling-like system, and the simplicity and small size of the current system.

E. Further Evaluation

The evaluations carried out so far have focused on memory usage and performance. Another important characteristic of sensor node applications is power consumption. Further measurements could be taken to compare the Insense VM's power consumption with other systems, and to reduce it if necessary.

VII. Conclusions

The Insense VM is a specialised JVM for running Insense programs on sensor nodes. Insense programs are compiled to Java, and then to bytecode using the standard Java toolchain. Using a split VM architecture, the classes are linked together on a more powerful machine, into a single binary which the VM can execute. This can be smaller than the original class files by an order of magnitude or more. The VM uses native methods to provide access to the underlying functionality of InceOS.

This work has aimed to demonstrate that a virtual machine is a viable and useful way to execute Insense programs, and that the JVM architecture, in particular, can be adapted to the needs of highly constrained devices. Measurements have shown that the VM fits within the memory limits of a sensor node, and that the performance overhead of interpretation is negligible for realistic wireless sensor network programs. Thus the benefits of a virtual machine – flexibility, portability, robustness, and the potential for easy over-the-air reprogramming – can be gained without compromising the usefulness of the language.

References

[1] D. M. Antonioli and M. Pilz, "Analysis of the Java class file format," University of Zurich, Tech. Rep., 1998.

[2] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom, "Introducing TakaTuka: a Java Virtual Machine for motes," in *Proceedings of the 6th ACM conference on Embedded Network Sensor Systems*. ACM, 2008, pp. 399–400.

[3] D. Balasubramaniam, A. Dearle, and R. Morrison, "A composition-based approach to the construction and dynamic reconfiguration of wireless sensor network applications," in *Software Composition*. Springer, 2008, pp. 206–214.

[4] N. Brouwers, P. Corke, and K. Langendoen, "Darjeeling, a Java compatible virtual machine for microcontrollers," in *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*. ACM, 2008, pp. 18–23.

[5] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich VM for the resource poor," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009, pp. 169–182.

[6] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, "Mote Runner: a multi-language virtual machine for small embedded devices," in *Third International Conference on Sensor Technologies and Applications, 2009 (SENSORCOMM '09)*. IEEE, 2009, pp. 117–125.

[7] "Contiki operating system," http://www.sics.se/contiki/.

[8] N. Costa, A. Pereira, and C. Serôdio, "A Java software stack for resource poor sensor nodes: towards peer-to-peer Jini," in *Embedded and Multimedia Computing, 2009 (EM-Com 2009)*. IEEE, 2009, pp. 1–6.

[9] A. Dearle, D. Balasubramaniam, J. Lewis, and R. Morrison, "A component-based model and language for wireless sensor network applications," in *Computer Software and Applications, 2008*. IEEE, 2008, pp. 1303–1308.

[10] P. Harvey, "InceOS: The Insense-specific operating system," Master's thesis, University of Glasgow, 2010.

[11] P. Harvey, A. Dearle, J. Lewis, and J. Sventek, "Channel and active component abstractions for WSN programming: a language model with operating system support," in *SensorNets 2012*, 2012.

[12] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Springer, 2005, pp. 115–148.

[13] J. Lewis and A. Dearle, "High-level abstractions for programming self-managing wireless sensor network applications," University of St Andrews, Tech. Rep., 2011.

[14] J. N. Lin and J. L. Huang, "A virtual machine-based programming environment for rapid sensor application development," in *Computer Software and Applications Conference, 2007 (COMPSAC 2007)*, vol. 2. IEEE, 2007, pp. 87–95.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013.

[16] W. Pugh, "Compressing Java class files," in *ACM SIGPLAN Notices*, vol. 34, no. 5. ACM, 1999, pp. 247–258.

[17] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: stack versus registers," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 4, pp. 2:1–2:36, 2008.

[18] 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 *Proceedings of the 2nd International Conference on Virtual Execution Environments*. ACM, 2006, pp. 78–88.