IEEE | IEEE Embedded Systems Letters

Making sensor node virtual machines work for real-world applications

Journal:	*Embedded Systems Letters*
Manuscript ID	IEEE-ESL-Mar-18-0036.R1
Manuscript Type:	Original Manuscripts
Date Submitted by the Author:	n/a
Complete List of Authors:	Reijers, Niels; National Taiwan University, Department of Computer Science and Information Engineering Ellul, Joshua; University of Malta, Department of Computer Science, Faculty of ICT Shih, Chi-Sheng; National Taiwan University, Computer Science
Keywords:	wireless sensor nodes, Virtual Machines, JVM

Making sensor node virtual machines work for real-world applications

Niels Reijers*, Joshua Ellul†, Chi-Sheng Shih*
*National Taiwan University †University of Malta

***Abstract*—A large number of Virtual Machines have been developed for sensor nodes. Many are based on Java, although implementations based on Python and .Net also exist. Some aspects of the JVM, such as the simple yet sufficiently expressive instruction set, work well on a sensor node. However, Java was developed for more powerful devices which results in certain functionality not being ideal for resource constrained devices.**

While writing virtual machine implementations and programmes to run on sensor node devices over the past decade, we encountered a number of situations where this mismatch between standard languages and the specifics of sensor node programming limited the practical use of current VMs. In this paper we describe what we consider to be the most pressing issues, and discuss how these may be resolved in future VMs.

***Index Terms*—Wireless Sensor Networks, Virtual Machines, Java, Embedded Systems.**

I. Introduction & Related Work

There are many advantages to using a Virtual Machine (VM) on sensor nodes or other Internet of Things (IoT) devices. IoT systems are expected to consist of heterogeneous platforms and architectures, and a VM allows for the use of the same binary to programme different underlying architectures. The use of high level languages can also speed up development in an environment where bare metal C programming is still common. Since the first VM, Maté [1], was presented for wireless sensor networks (WSNs), a wide range of other resource constrained VMs have been proposed, supporting languages like Java [1] [2]–[4], Python [2] or .Net [5, 6]. Due to space constraints we cannot discuss each individually, but refer to the respective papers and [7] for an overview.

IoT devices come in a wide range. Devices like the Raspberry Pi can run a full OS stack, but tiny sensor nodes have remained popular for two decades because their resource constrained design allows them to scale down to even smaller sizes, and significantly lower power consumption. However, this design also poses specific challenges. Sensor nodes often separate code in flash memory, and mutable data in RAM. Since RAM needs power to maintain its state even when the node is in sleep mode, it is often limited to only a few KB. This means efficient use of RAM is very important on sensor nodes. Flash memory, though larger at tens to a few hundred KB, is also a scarce resource. Many sensing tasks are time bounded, so (predictable) performance is important. Finally, performance is directly related to CPU active time, and thus to power consumption. Since the main reason to use resource constrained devices is their extremely low power use, the fact that interpreting VMs are one to two orders of magnitudes slower than native code is a problem in cases that require more computation. However, Ahead-of-Time (AOT) compilation can improve performance to roughly half of native code [4, 8].

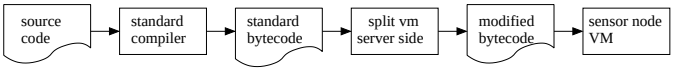

Fig. 1. Typical sensor node split VM architecture

Whether based on Java, .NET, or Python, it is common practice to support only a (substantial) subset of the original language features and modify the bytecode to some extent, to make it more suitable to run on a sensor node. Typically, a 'split VM' architecture is used, where the bytecode generated by the standard compiler is first processed by a desktop component of the VM before being sent to the node, as shown in Figure 1. Common changes include: (i) removing method- and classnames to reduce code size; (ii) dropping support for reflection; (iii) adding 16-bit operations to reduce memory consumption and execution overhead; (iv) separating references from integers for easier garbage collection; and (v) removing constant pool look-ups where possible.

Such changes are limited, since the VMs are still bound by the source language. Having used and developed sensor node VMs for several years, we encountered a number of issues caused by a mismatch between language features, and the very specific characteristics of sensor node code. While previous work has shown that VMs are feasible on a sensor node, we observe that in order to make them work *well* in real-world applications, some sensor node specific changes are necessary.

In this paper we describe the most important issues, affecting many aspects of software development, from memory usage, which is a critical resource on sensor nodes, to making sure code is readable and easy to maintain. For each issue we propose options on how they could be resolved or improved.

II. Improving current sensor node VMs

In this section we discuss the most pressing issues with current sensor node VMs, summarised in Table I. Where possible we quantify their impact using five benchmarks with typical sensor node code: the standard CoreMark [3] benchmark,

This research was supported in part by the Ministry of Science and Technology of Taiwan (MOST 105-2633-E-002-001), National Taiwan University (NTU-105R104045), Intel Corporation, and Delta Electronics.

[1] http://harbaum.org/till/nanovm/index.shtml
[2] http://code.google.com/p/python-on-a-chip/
[3] http://www.eembc.org/coremark/index.php

TABLE I
POINTS REQUIRING ATTENTION IN FUTURE SENSOR NODE VMS AND THEIR RELATED OVERHEAD

Sec.	Issue	in	affects	Evaluation	CoreMk	MoteT.	HeatD.	LEC	FFT8	FFT16
II-A	A tailored standard library	Standard library	VM size							
II-B	Support for constant data	Java lang., JVM	memory usage, code size	Const array RAM overhead		cannot		67 B	264 B	2 KB
				Const array flash overhead		fit		135 B	1.4 KB	6 KB
II-C	Better lang. support for shorts and bytes	Java lang.	memory usage, source maintainability	Casts per 100 LOC	8	7	24	13	29	22
II-D	Simple type definitions	Java lang.	source maintainability							
II-E	Explicit and efficient inlining	Java lang.	performance	Slowdown non-inlined version	13%				45%	20%
				Size non-inlined version	+48 B				-50 B	-22 B
II-F	An optimising compiler	javac	performance	Slowdown w/o optimisations	117%	75%	2%		16%	3%
II-G	Allocating objects on stack	Java lang., JVM	(predictable) performance	Slowdown heap allocation	6%	65%		330%		
II-H	Reconsidering adv. language features OO, GC, threads, exceptions	Java lang., JVM	VM size, complexity, and performance							

MoteTrack [4] indoor localisation, an application from our own group that detects heat sources using an 8x8 pixel heat sensor, LEC lossless compression for sensor nodes [9], and 8 and 16-bit FFT using the common `fix_fft.c` implementation.

A. A tailored standard library

A minimum Java API for resource constrained devices was proposed by Sun Microsystems, namely the Connected Limited Device Configuration (CLDC) specification [10]. This was primarily intended for devices larger than resource constrained sensor nodes and providing full support would require a substantial amount of memory for features that are rarely required for sensor node applications. Table II shows the code size for library support in the Darjeeling VM [2].

Aslam [3] presents a method for dead code removal that could be used to remove unused code from a library. However, such code becomes part of the application that is uploaded to a device, which is less efficient than a native implementation, and not possible for library functions that access hardware. Therefore we argue that a minimal, natively implemented and efficient library is necessary that is present on all devices.

Below we outline the most notable features of the CLDC specification that are not necessary for most WSN applications.

String support, which requires a substantial amount of program space, is rarely required within sensor node applications. Communication with peripheral hardware is often by means of a stream of bytes. To facilitate such code it would be of benefit to support constant strings at the programming language level (which could be converted to byte arrays under the hood).

A CLDC `Stream` abstraction is intended to facilitate file, network and memory operations. The abstraction is not well suited for communication protocols required by WSN applications such as I^2C and SPI. In CLDC, connections between devices can be initiated by specifying URI-like strings. On the other hand, WSN nodes identify other nodes using radio transceiver IDs, typically a 16 or 32-bit identifier.

It is clear from the specification that WSN applications were not the intended target. We argue that a tailored library should be designed specifically for sensor node applications. Such a library would include functionality for: (i) basic math; (ii) array operations; (iii) a communication API that encapsulates the low-level protocols typically used (e.g. I^2C); and (iv) a higher-level generic radio and sensor API abstraction.

[4] http://www.eecs.harvard.edu/~konrad/projects/motetrack

B. Support for constant data

While Java allows us to declare variables as `final`, this is only a language level feature, and the VM has no concept of constant data. This is not surprising, since most physical CPUs do not make the distinction either. However, this is different on a sensor node's Harvard architecture where code and data memory are split. The amount of flash memory is usually several times larger than the available RAM, so constant data should be kept in flash instead of wasting precious RAM on data that never changes. This is especially important for arrays of constant data, which are common in WSN applications.

When we implement them as a `final` Java array, the compiler emits a static class initialiser that creates a Java array object, and then uses the normal array access instructions to initialise each element individually:

```
sspush(256);newarray;          // create the array
adup;sconst_0;sconst_0;bastore;// set index 0
adup;sconst_1;sconst_3;bastore;// set index 1, etc..
```

There are two problems with this: (i) the array will occupy scarce RAM; and (ii) initialising array elements using JVM instructions requires 4 instructions per element, resulting a large code size overhead, especially when AOT compilation is used. In our benchmarks we see the constant sine wave table in the 16-bit FFT wastes 2 KB of RAM, and MoteTrack, which has a 20 KB database of reference data, cannot run without special optimisations to put this in flash.

Possible solutions Supporting constant arrays will require changes to both the language and bytecode. From the programmer's perspective it should be enough to simply add an annotation like '`progmem`' to an array declaration to tell the compiler it should be stored in flash. The bytecode will then need to be expanded to allow constant arrays in the constant pool, and to add new versions of the array load instructions (e.g., `AALOAD` and `IALOAD`) to read from them.

C. Better language support for shorts and bytes

Because RAM is scarce, 16-bit short and single byte data types are commonly used in WSN code. The standard JVM only has 32 and 64-bit operations, and variables (instance, local or static) and stack values are stored as 32-bit, even if the actual type is shorter. On a sensor node this wastes memory, and causes a performance overhead since most nodes have 8 or 16-bit architectures, so many sensor node JVMs introduce 16-bit operations and store values in 16-bit slots [2].

However, redesigning the VM is only half of the solution. At the language level, Java defines that an expression evaluates to 32-bits, or 64-bits if at least one operand is long. Attempting to store this in a 16-bit variable will result in a 'lossy conversion' error at compile time.

As an example, if we have 3 `short` variables, a, b, and c, and want to do `a=b+c;`, we need to insert a cast to avoid errors from the Java compiler: `a=(short)(b+c);`

Also, passing literal integer values to a method call treats them as ints, even if they are short enough to fit in a smaller type, which results in calls like: `f((byte)1);` These casts, up to 29 per 100 lines of code in our benchmarks, can make the code much harder to read.

Possible solutions We suggest that C-style automatic narrowing conversions would make most sensor node code more readable, but to leave the option of Java's default behaviour open, we may implement this as new datatypes: Declaring variable `a` as `unchecked short` would implictly narrow to short when needed, so `a=b+c;` would not need an explicit cast anymore, while declaring it as a normal `short` would.

D. Simple type definitions

On a sensor node, the limited resources force us to adopt different design patterns compared to desktop software. In normal Java we usually rely on objects for type safety and to keep code readable and easy to maintain. On sensor nodes, objects are expensive and we frequently make use of simple integer variables for a multitude of different tasks for which we would traditionally use objects.

Such cases often result in many integer type variables, and we found that our code would be easier to maintain if we could create aliases for the `int`, `short` or `byte` integer types to explicitly indicate their meaning. Type checking on these types would also add a welcome layer of extra safety.

Possible solutions At a minimum, we should have a way to define simple aliases for primitive types, similar to C's `typedef`. A more advanced option that fits more naturally with Java, would be to have a strict `typedef` which also does type checking, so that a value of one user defined integer type cannot be accidentally assigned to a variable of another type, without an explicit cast.

E. Explicit and efficient inlining

Java method calls are inherently more expensive than C functions. We found this often is a problem for small helper functions that are frequently called. Tools like Proguard[5] can be used to inline small methods, but only if they are smaller than Proguard's threshold. This leaves developers with two unattractive options for larger but performance critical methods: either leaving it as a method and accepting the performance penalty, or manually copy-pasting the code, which is error-prone and leads to code that is harder to maintain.

In our benchmarks we use the method call optimisation described in [4], without which the overhead would be several times larger, but still see a slowdown of up to 45%, while the size increase of inlining is very modest. In one case inlining a very small method reduced code size because the method call and test on the return value were larger than the inlined version.

TABLE II
SIZE OF DARJEELING VM COMPONENTS

Component	std.lib (bytes)	VM (bytes)	total (bytes)
core vm	3529	7006	10535
strings	8467	1942	10409
interpreter loop	0	10370	10370
gc	80	3442	3522
threads	909	2472	3381
exceptions	1338	818	2156
math	222	1274	1496
io	530	680	1210
total	15075	28004	43079

Possible solutions Developers should be given more control over inlining, which could be achieved by an `inline` keyword to force the compiler to inline important methods.

F. An optimising compiler

Java compilers typically do not optimise the bytecode and translate the source more or less as-is. Without a clear performance model it is not always clear which option is faster, and the bytecode is expected to be run by a JIT compiler, which can make better optimisation decisions knowing the target platform and runtime behaviour. However, a sensor node does not have the resources for this and must execute the code as it is received. This leads to significant overhead, for example by repeatedly reevaluating a constant expression in a loop.

Possible solutions Even without a clear performance model, some basic optimisations can be done. To produce better bytecode, we applied the simple manual optimisations described in [4], which leads to code up to twice as fast.

G. Allocating objects on stack

In Java anything larger than a primitive value has to be allocated on the heap. This introduces a performance overhead, both for allocating the objects, and the occasional garbage collection run, which may take several thousand cycles, are a problem for code with specific timing constraints.

The LEC benchmark uses a small object to pass two return values from a method. The natural way to do this is to create an instance where it's needed, but the overhead from frequent allocations can be significant. The alternative is allocating such objects only once, at an earlier stage, and passing them down to where they are needed, potentially through multiple methods. This can be up to 4x faster, but leads to messy code.

Possible solutions On desktop JVMs escape analysis [11] is used to determine if an object can be safely allocated on the stack. While this is too complex for a sensor node, it could be done offline, similar to TakaTuka's offline garbage collector analysis [3]. The bytecode can be extended with new versions of the `new` opcodes to place an object in the stack frame instead of the heap.

There is a risk to doing this automatically. In many sensor node VMs, the split between heap and stack memory is fixed, and both are limited. If the compiler automatically puts all

[5]http://www.guardsquare.com/en/proguard

objects that could be on the stack in the stack frame instead of the heap, we may end up with an empty heap, and a stack overflow. Therefore, it is better to leave this optimisation to the developer by also introducing a new keyword at the language level, and using escape analysis to verify this keyword is only used in places where the object can be allocated on the stack.

H. Reconsidering advanced language features

Finally, we conclude with some discussion on more fundamental language design choices. Many tiny VMs implement some of Java's more advanced features, but we are not convinced these are a good choice on a sensor node.

While useful, these features come at a cost. The trade-off when writing sensor node code is significantly different: many of these features are vital to large-scale software development, but the size of sensor nodes programmes is much smaller. And while VM size is not an issue on the desktop, these features are relatively expensive to implement on a sensor node.

In Table II we show the code size for the features we discuss below. Combined, the features below and the string functions mentioned in Section II-A make up about half the VM.

These features also cause a performance penalty, and some are much harder in an AOT compiler where we can't implement them in the interpreter loop. This means that if we care about performance and the corresponding reduction in CPU energy consumption, we either have to give them up, or spend considerably more in terms of VM complexity and size.

1) Threads: As shown in Table II, support for threads costs about 10% of the VM size, if we exclude the string library. In addition, a stack is needed for each thread. If we allocate a fixed block, it must be large enough to avoid stack overflows, but too large a block wastes precious RAM. Darjeeling allocates stack frame as a linked list on the heap. This is memory efficient, but allocating on the heap is slower and occasionally triggers the GC. We therefore argue a cooperative concurrency model is more appropriate where lightweight threads voluntarily yield the CPU and share a single stack.

2) Exceptions: In terms of code size, exceptions are not very expensive to implement in an interpreter, but again, they are harder to implement in an AOT compiler. We also feel the advantage of having exceptions is much lower than the other features mentioned in this section. They could be easily replaced with return values to signal errors.

3) Virtual methods: It is hard to quantify the overhead of implementing virtual methods since the code for handling them is integrated into several functions. In terms of size it is most likely less than 2KB, but the overhead for resolving a virtual method call is considerable, and an AOT compiler can generate much more efficient code for static calls.

In practice we seldom used virtual methods in sensor node code, but some form of indirect calls is necessary for things like signal handling. It should be possible to develop a form of function pointers that can be implemented more efficiently. However, the details will require more careful study.

4) Garbage collection: Finally, garbage collection is clearly the most intrusive one to change. While the first three features could be changed with minor modifications to Java, the managed heap is at its very core.

Still, there are good reasons for considering alternatives. Table II shows the GC methods in Darjeeling add up to about 3.5KB, but the actual cost is much higher as many other parts of Darjeeling are influenced by the garbage collector.

Specifically, it is the reason Darjeeling uses a 'split-stack' architecture: the operand stack and variables are split into a reference and integer part. This makes it easy for the GC to find live references, but leads to significant code duplication and complexity. When using AOT compilation, the split stack adds overhead to maintain this state, and the extra register we have to reserve as a second stack pointer.

III. Conclusion and future work

In this paper we described a number of issues we encountered over the years while using and developing sensor node VMs. They may not apply to every scenario, but the wide range of the issues we present suggests many applications will be affected by at least some.

Most sensor node VMs already modify the instruction set of the original VM and usually support only a subset of the original language. The issues described here indicate these changes do not go far enough, and we still need to refine our VMs further to make them truly useful in real-world projects.

There are two possible paths to follow: a number of issues can be solved by improving existing Java-based VMs. Staying close to Java also has the advantage of being able to reuse existing knowledge and infrastructure.

However the issues discussed in the last few sections require more invasive changes to both the source language and VM. If the goal is to run platform independent code safely and efficiently, rather than running Java, we should start from the specific requirements and constraints of sensor node software development. We suspect this would lead us to more lightweight features and more predictable memory models.

For either path, we hope the points presented in this paper can help in the development better future sensor node VMs.

References

[1] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *ACM Sigplan Notices*, vol. 37, no. 10. ACM, 2002.
[2] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, A Feature-rich VM for the Resource Poor," in *SENSYS*, 2009.
[3] F. Aslam, "Challenges and Solutions in the Design of a Java Virtual Machine for Resource Constrained Microcontrollers," Ph.D. dissertation, University of Freiburg, 2011.
[4] N. Reijers and C.-S. Shih, "Improved Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices," 2017, arXiv:1712.05590.
[5] Y. Kishino, Y. Yanagisawa, T. Terada, M. Tsukamoto, and T. Suyama, "CILIX: a Small CIL Virtual Machine for Wireless Sensor Devices," in *Pervasive*, 2010.
[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 *SENSORCOMM*, 2009.
[7] N. Costa, A. Pereira, and C. Serodio, "Virtual Machines Applied to WSN's: The state-of-the-art and classification," *ICSNC '07*.
[8] J. Ellul and K. Martinez, "Run-Time Compilation of Bytecode in Sensor Networks," in *SENSORCOMM*, 2010.
[9] F. Marcelloni and M. Vecchio, "An Efficient Lossless Compression Algorithm for Tiny Nodes of Monitoring Wireless Sensor Networks," *The Computer Journal*, vol. 52, no. 8, Nov. 2009.
[10] Sun Microsystems, Inc., "Connected Limited Device Configuration Specification Version 1.1," March 2003.
[11] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape Analysis for Java," *OOPSLA '99*.