國立臺灣大學電機資訊學院資訊工程學系

博士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Doctoral Dissertation

開普敦 VM

CapeVM: A Fast and Safe Virtual Machine for Resource-Constrained Internet-of-Things Devices

賴爾思
Niels Reijers

指導教授：施吉昇教授
Advisor: Professor Chi-Sheng Shih

中華民國 107 年 3 月
March, 2018

國立臺灣大學博士學位論文
口試委員會審定書

開普敦 VM
CapeVM: A Fast and Safe Virtual Machine for Resource-Constrained Internet-of-Things Devices

本論文係賴爾思君 (D00922039) 在國立臺灣大學資訊工程學系完成之博士學位論文，於民國 107 年 3 月 28 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

_____

_____ _____

_____ _____

_____ _____

_____ _____

所　長：_____

Acknowledgements

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.

摘要

中文摘要

關鍵字： 關鍵字

Abstract

Many virtual machines have been developed targeting resource-constrained sensor nodes. While packing an impressive set of features into a very limited space, most fall short in two key aspects: performance, and a safe, sandboxed execution environment. Since most existing VMs are interpreters, a slow-down of one to two orders of magnitude is common. Given the limited resources available, verification of the bytecode is typically omitted, leaving them vulnerable to a wide range of possible attacks.

In this dissertation we propose CapeVM, a sensor node VM aimed at delivering both high performance and a sandboxed execution environment that guarantees malicious code cannot corrupt the VM's internal state or perform actions not allowed by the VM.

CapeVM uses Ahead-of-Time compilation to native code to improve performance and introduces a range of optimisations to eliminate most of the overhead present in previous work on sensor node AOT compilers. A safe execution environment is guaranteed by a set of run-time and translation-time checks. The simplicity of the VM's instruction set allows us to perform most of these checks when the bytecode is translated to native code, reducing the need for expensive run-time checks compared to native code approaches.

We evaluate CapeVM using a set of 12 benchmarks with varying characteristic, including the commercial *CoreMark* benchmark and a number of real sensor node applications. While some overhead from using a VM and added safety checks cannot be avoided, the evaluation shows CapeVM's optimisa-

tions reduce this overhead dramatically. This results in a performance 2.1x slower than unsafe native code, which is comparable to or better than existing native solutions to provide safety. Without safety checks, the overhead drops to 1.7x. Thus, CapeVM combines the desirable properties of existing work on both safety and virtual machines for sensor networks with significantly improved performance.

Keywords: wireless sensor networks, Internet of Things, Java, virtual machines, ahead-of-time compilation, software fault isolation

Contents

List of Figures

List of Tables

List of Listings

Table 1: Effect of constant arrays on size and performance

	RC5		FFT		LEC		MoteTrack	
Size of constant data	200		2,048		51		20,560	
Using constant arrays	no	yes	no	yes	no	yes	no	yes
Performance overhead	19.5%	19.5%	17.8%	17.7%	86.5%	84.7%	cannot run	156.3%
Size of constant data in flash	1998	204	26,714	2,052	930	59	cannot run	20,588
Size of constant data in RAM	208	0	2,056	0	67	0	cannot run	0

0.1 Constant arrays

Four benchmarks contain arrays of constant data, which were stored in flash memory by placing them in classes with the `@ConstArray` annotation. To evaluate the effect of this optimisation, we compare them to versions without this annotation. The results are shown in Table 1. There are three advantages to this optimisation: a small improvement in performance, reduced code size, and reduced memory usage.

When using constant arrays, the id of the array to read from is a bytecode parameter in the `GETCONSTARRAY` instruction. No reference to the array needs to be loaded on the stack, and the calculation to find the address of the target element is slightly easier, which results in a modest reduction in performance overhead of 1.8% for the *LEC* benchmark and 0.1% for *FFT*.

The real advantage however, is the reduction in code size and memory usage. Without this optimisation, an array of constant data is transformed into normal bytecode in the class initialiser that will create an array object on the heap and fill each element individually, as shown in Listing **??**.

The class initialiser uses four bytecode instructions per element to fill each element of the array. For an array of bytes, this can take up to 7 bytes of bytecode for each byte of data, which increases even further after AOT compilation. In the *LEC* benchmark this results in a class initialiser of 928 bytes, over *18 times* the size of the original data.

For such a small array this might still be acceptable, but the 26 KB needed to store *FFT*'s 2 KB of data is a significant overhead, and while *MoteTrack*'s 20 KB of data could fit in flash memory, the resulting class initialiser cannot. When using the constant array optimisation, the array is stored as raw data in the constant pool, resulting in just 4 bytes

of overhead per array.

The final, and most significant advantage of this optimisation is that the array is no longer stored in RAM. Again, the 67 bytes of RAM needed to store *LEC*'s two constant arrays, each with 8 bytes overhead for the heap and array headers, may be acceptable. For *RC5* the 208 byte RAM overhead is starting to be significant, and while the *FFT* benchmark can still run without the constant array optimisation, its array consumes over half of the ATmega128's 4 KB of RAM. For *MoteTrack*, the size of its constant arrays is well over the amount of RAM available, making it impossible to run this benchmark without the constant array optimisation.

Chapter 1

Lessons from JVM

In Section **??** we defined two of our main research questions as how close an AOT compiling sensor node VM can come to native performance, and whether a VM is an efficient way to provide a safe execution environment. These questions are not specific to Java, and the main motivation to base CapeVM on Java was the availability of a rich set of tools and infrastructure to build on, including a solid VM to start from in the form of Darjeeling.

One aspect of the JVM that makes it an attractive choice for sensor nodes is its simplicity, allowing a useful subset of it to be implement in as little as 8 KB [7]. However, it also lacks some important features that make it a less good fit for typical sensor node code. These range from minor annoyances that reduce code readability, to the lack of support for constant data and high memory consumption for nested data structures. The last two issues make some applications that can run on a sensor node when written in C, impossible to implement in Java.

In this chapter we discuss the most pressing issues we encountered, summarised in Table 1.1, and suggest ways they could be improved in future VMs. Where possible, the impact of these issues is quantified in Table 1.2.

Although more study is required to turn these suggestions into working solutions, many of the points raised here could be improved with minor changes to Java, leading to a 'sensor node Java', much like nesC [3] is a sensor node version of C, but some require more drastic changes.

Table 1.1: Point requiring attention in future sensor node VMs

Section	Issue	in	affects
1.1	A tailored standard library	Standard library	VM size
1.2	Support for constant arrays	Source language, VM	memory usage, code size
1.3	Support for nested data structures	Source language, VM	memory usage, performance
1.4	Better lang. support for shorts and bytes	Source language	memory usage, source maintainability
1.5	Simple type definitions	Source language	source maintainability
1.6	Explicit and efficient inlining	Source language	performance
1.7	An optimising compiler	Compiler	performance
1.8	Allocating objects on stack	Source language, VM	(predictable) performance
1.9	Reconsidering adv. language features threads, exceptions, OO, garbage collection	Source language, VM	VM size, complexity, and performance

1.1 A tailored standard library

A minimum Java APIs for resource-constrained devices, the Connected Limited Device Configuration (CLDC) specification, was proposed by Sun Microsystems [9]. The CLDC was primarily intended for devices larger than typical sensor nodes, and not tailored to the characteristics of typical sensor node code. Providing support for the full CLDC specification would require a substantial amount of memory and programme space for features that are rarely required by sensor node applications. Table 1.3 shows the code size of library support as implemented in the original Darjeeling VM.

The largest mismatch comes from the CLDC's string support, which takes up over 8 KB. While string support is one of the most basic features one would expect to find in the standard library of any general purpose language, it is rarely required within sensor node applications that usually do not have a UI and only communicate with the outside world through radio messages.

On the other hand, the standard library should include abstractions for typical sensor node operations that are missing from the CLDC. The 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. However, processing these is relatively expensive, and WSN nodes often identify other nodes using a 16 or 32-bit identifier.

Aslam [1] discusses a method for dead code removal that could be used to remove unused code from a library. The remaining library code becomes part of the application that is uploaded to a device as a whole. While this can be useful to allow developers to use a large library of seldom used functions that will only be included when needed, this is much less efficient compared to a natively implemented standard library, and not possible for library functions to access the hardware.

Therefore we argue a minimal tailored library is necessary that may be efficiently implemented in native code and present on all devices, and that this library should be designed from the ground-up 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.

Table 1.2: Quantitative impact of Java/JVM issues

Section	Measure [a]	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect
1.2	Size of constant data						200	2,048		51		20,560		
	Const array RAM overhead						208	2,056		67		too big		
	Const array flash overhead						1,998	26,714		930		too big		
1.3	Size of main data structures in C	512	512	200	144	174	256	256	860	1024	1633 [b]	606	644	1088
	Size of main data structures in Java	520	520	208	160	214	288	272	884	1058	1983	1387 [c]	676	1158
	Size increase	1.6%	1.6%	4.0%	11.1%	23.0%	12.5%	6.3%	2.8%	3.3%	21.4%	128.9%	5.0%	6.4%
1.4	Casts	1	6	5	8	8	8	16	3	8	41	33	4	70
	Lines of code [d]	11	24	16	38	165	27	73	43	48	637	475	47	272
	Casts per 100 LOC	9	25	31	21	5	30	22	7	17	6	7	9	26
1.6	Slowdown non-inlined version		69%		57%	25%	37%	20%			8%			
	Size non-inlined version		+42		-224	-1502	-94	-20			+48			
1.7	Slowdown w/o optimisations	91%	52%	544%	3%			3%	23%		116%	76%		2%
1.8	Slowdown heap allocation									330%	6%	65%		

[a] A blank entry indicates the benchmark was not affected. Highlights indicate a significant impact.
[b] Actual amount of memory used. CoreMark's C version allocates 2047 bytes, but the remaining space is not used.
[c] After replacing Motetrack's 2-byte RSSI array with two variables.
[d] Counted as the number of actual code lines, excluding blanks lines, comments, and single brackets.

Table 1.3: 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
Garbage collection	80	3442	3522
Threads	909	2472	3381
Exceptions	1338	818	2156
Math	222	1274	1496
IO	530	680	1210
Total	15075	28004	43079

1.2 Support for constant arrays

Constant data is relatively common in sensor node code. In our benchmarks, they appear as the key schedule in the *RC5* cipher, a table of precomputed sine wave values for the *FFT* benchmark, a dictionary of codes in the *LEC* benchmark, and a database of RSSI signatures in *MoteTrack*.

Sensor node CPUs differ from desktop systems in the fact that memory is split in a small amount of RAM for volatile data, and a relatively large amount of flash memory for code and constant data. Because Java was not designed for such systems, it has no way to distinguish between the two, so both constant and variable data are always placed in RAM.

There are two problems with Java's approach: (i) an array of constant data will take up RAM, which is a scarce resource, and (ii) the data is not stored as raw data, but as a sequence of bytecode instructions that initialise each element of an array individually. In the worst case, an array of bytes, this means 7 byte of bytecode are needed for each byte of data, which increases even further after AOT compilation.

An extension that allows developers to place arrays of constant data in flash memory was presented in Section **??**.

1.3 Support for nested data structures

Besides the need to support constant data, the *MoteTrack* benchmark also exposes another weakness of Java: it does not support data structures of many small objects efficiently.

Listing 1 shows the main `RefSignature` data structure used in *MoteTrack*. This structure consists of a location, which is a simple struct of 3 shorts, and a signature, which has an id, and an array of 18 signals. A signal is defined by a source ID, and an array of 2 elements with RSSI values.

```
1   #define NBR_RFSIGNALS_IN_SIGNATURE 18
2   #define NBR_FREQCHANNELS            2
3
4   struct RefSignature
5   {
6       Point location;
7       Signature sig;
8   };
9
10  struct Point
11  {
12      uint16_t x;
13      uint16_t y;
14      uint16_t z;
15  };
16
17  struct Signature
18  {
19      uint16_t id;
20      RFSignal rfSignals[NBR_RFSIGNALS_IN_SIGNATURE];
21  };
22
23  struct RFSignal
24  {
25      uint16_t sourceID;
26      uint8_t rssi[NBR_FREQCHANNELS];
27  };
```

Listing 1: MoteTrack `RefSignature` data structure

Since all the arrays are of fixed length, in C the layout of the whole structure is known at compile time, shown in Figure 1.1. As described in Section **??**, in Java every object is made up of a list of primitive values: either an int or a reference to another object. In Java we cannot have an array of objects, only an array of *references to* objects. Thus, the most natural way to translate the C structures in Listing 1 to Java, is as a collection of objects and arrays on the heap, as shown in the right half of Figure 1.1. Note that every one of the 18 `RFSignal` structs becomes an object, which in turn has a pointer to an array of RSSI values.

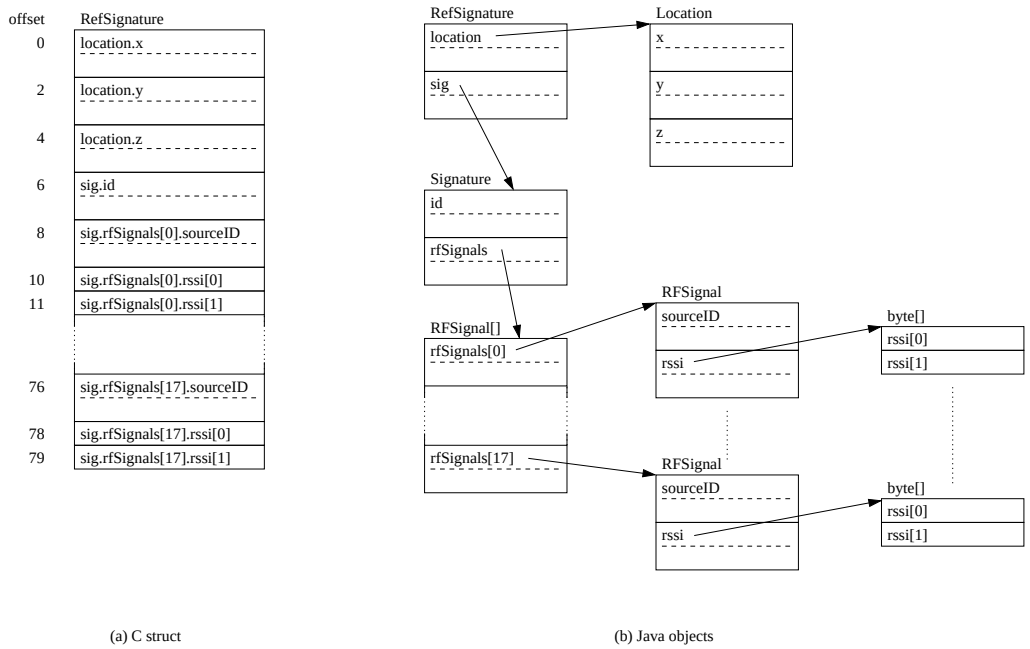

Figure 1.1: The `RefSignature` data structure: as a C struct, and as a collection of Java objects

There are two problems with this. First, since the location of these Java objects is not known until run time, there is a performance penalty for having to follow the chain of references. *MoteTrack* will loop over the signals in the `rfSignals` array. Starting from this array, Java needs to do 3 lookups to get to the right RSSI value: the address of the current `RFSignal` object, the address of the `rssi` array, and then the actual RSSI value. For the C version, all the offsets are known at compile time, so the compiler can generate a much more efficient loop, directly reading from the right locations.

The second problem is the added memory usage. The C struct only takes up 80 bytes, all used to store data. The Java version allocates a total of 40 objects, 36 of which are spent on the `RFSignal` objects and their arrays of RSSI values. Each of these requires a heap header, which takes up 5 bytes. In addition, the 18 byte arrays have a 3 byte header and the signal array a 4 byte header, and the whole structure contains a total of 39 references, which each take up 2 bytes. In total, the collection of Java objects use $80 + 40 * 5 + 18 * 3 + 4 + 39 * 2 = 416$ bytes.

Combined with *MoteTrack*'s other data structures, this is too large to fit in memory, which forced us to refactor the 2 element `rssi` array into two byte variable stored directly in `RFSignal`, as explained in Section **??**. This allowed us to run the benchmark, but a

RefSignature still takes up 236 bytes, and reading a RSSI value still takes two lookups instead of one.

Generally, large arrays of primitive types do not suffer from this problem and can be stored efficiently, but for programmes containing large arrays of objects this causes significant overhead. In some cases we can work around this problem by flattening the structure, for instance in *MoteTrack*'s case we could replace the array of `RFSignals` with three separate arrays for `sourceIDs`, `rssi_0` and `rssi_1`, but only at a significant cost in readability.

1.4 Better language support for shorts and bytes

Because RAM is scarce, 16-bit short and single byte data types are commonly used in sensor node code. The standard JVM only has 32 and 64-bit operations, and variables 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-bit or 16-bit architectures. Therefore, many sensor node JVMs, including Darjeeling, introduce 16-bit operations and store values in 16-bit slots.

However, this is only one 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 a **`long`**. Attempting to store this in a 16-bit variable will result in a 'lossy conversion' error at compile time, unless explicitly cast to a **`short`**.

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);
```

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);
```

While seemingly a small annoyance, in more complex code that frequently uses of shorts and bytes, these casts can make the code much harder to read. Table 1.2 shows that over 25 casts per 100 lines of code can appear in some benchmarks.

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, while it would if `a` is declared as a normal `short`.

1.5 Simple type definitions

When developing code for a sensor node, the limited resources often result in different design patterns compared to desktop software. In normal Java code we usually rely on objects for type safety and keeping code readable and easy to maintain. On sensor nodes, objects are expensive and we frequently make use of shorts and ints for a multitude of different tasks for which we would traditionally use objects.

In these situations we often found that our code would be much easier to maintain if there was a way to name new integer types to explicitly indicate their meaning, instead of using many of `int` or `short` variables. Having type checking on these types would add a welcome layer of 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.

1.6 Explicit and efficient inlining

Java method calls are inherently more expensive than C functions. On the desktop, JIT compilers can remove much of this overhead, but a sensor node does not have the resources for this. We often found this to be a problem for small helper functions that are frequently called. As an example, the C version of the *XXTEA* benchmark contains this macro:

```
1  #define MX (((z>>5^y<<2) + (y>>3^z<<4)) \\
2              ^ ((sum^y) + (key[(p&3)^e] ^ z)))
```

This macro is called in four places, and is very performance critical. Tools like Proguard [6] can be used to inline small methods, but `MX` is larger than Proguard's size threshold. This leaves developers with two unattractive options: 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.

Possible solutions The simplest solution would be to have a preprocessor similar to C's. However, such a low level text-based solution may not be the most user friendly solution for developers without a C background.

Another option is to give the developer more control over inlining, which could easily be achieved by adding an `inline` keyword to force the compiler to inline important methods.

1.7 An optimising compiler

As discussed in previous chapters, but listed here again for completeness, Java compilers typically do not optimise the bytecode but translate the source almost 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 run-time 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. The results in Section **??** show that some very conservative optimisations can already result in code twice as fast as the original. These could be further expanded,

and combining the tasks of the optimiser and infuser can further improve performance as shown in Section **??**.

1.8 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 run of the garbage collector that may take several thousand cycles.

In our benchmarks we encountered a number of situations where a temporary object was needed. For example, the `encode` function in the *LEC* benchmark needs to return two values: *bsi* and the number of bits in *bsi*. In C this is done by passing two pointers to `encode`. In Java we can wrap both values in a class and either create and return an object from `encode`, or let the caller create it and pass it as a parameter for `encode` to fill in.

In code that frequently needs short-lived objects the overhead for allocating them can be significant, and unpredictable garbage collector runs are a problem for code with specific timing constraints. Besides *LEC*, we saw similar situations in the *CoreMark*, *MoteTrack* and *Heat detection* benchmarks. The problem is especially serious on a sensor node, where the limited amount of memory means the garbage collector is triggered frequently even if relatively few temporary objects are created.

```
1   public static short LEC(short[] numbers, Stream stream) {
2       BSI bsi = new BSI();      // Allocate bsi only once
3       for (...) {
4           ...
5           compress(ri, ri_1, stream, bsi);
6           ...
7       }
8   }
9
10  private static void compress(short ri, short ri_1, Stream stream, BSI bsi) {
11      ...
12      encode(di, bsi);          // Pass bsi to encode to return both value and length
13      ...
14  }
15
16  private static void encode(short di, BSI bsi) {
17      ...
18      bsi.value = ...           // return value and length by setting object fields
19      bsi.length = ...
20  }
```

Listing 2: Avoiding multiple object allocations in the LEC benchmark

This overhead can often be reduced by allocating earlier and reusing the same objects in a loop. In Listing 2 we see this implemented for the *LEC* benchmark, where we use the `bsi` object to return two values from `encode` to `compress`. Instead of creating a new object in each iteration in the `compress` method where it is needed, we create `bsi` once, outside of the main loop, and pass the same object to `compress` multiple times.

This technique of pulling object creation up the call chain can often be used to remove this sort of overhead, and it worked in all four benchmarks mentioned before. However, it gets very cumbersome if the number of objects is more than one or two, or if they need to be passed through multiple layers. Readability is also reduced, since objects that are only needed in a specific location are now visible from a much larger scope.

Possible solutions On desktop JVMs, escape analysis [2, 4] is used to determine if an object can be safely allocated on the stack instead of the heap, thus saving both the cost of heap allocation, and the occasional garbage collection run triggered by it.

While the analysis of the bytecode required for this is far too complex for a sensor node, it could be done offline, similar to TakaTuka's offline garbage collector analysis. The bytecode can then be extended by adding special versions of the **`new`** opcodes to instruct the VM to place an object in the stack frame instead of the heap. A field should also be added to the method header to tell the VM how much extra space for stack objects needs to be reserved in the stack frame.

There is a risk to doing this automatically. In our sensor node VM, the split between heap and stack memory is fixed, and both are limited. If the compiler automatically puts all 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, so developers can explicitly indicate which objects go on the stack and which in the heap. Of course escape analysis is still necessary to check at compile time this keyword is only used in places where it is legal for the object to be allocated on the stack.

1.9 Reconsidering advanced language features

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

While features like threads and garbage collection are all useful, they come at a cost. The trade-off for a sensor node VM is significantly different from a desktop VM: many of Java's more advanced 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 with limited flash memory. We believe a VM developed from scratch, with the aim of providing platform independence, safety, and performance through AOT compilation, would end up with a design very different from the JVM.

Table 1.3 shows the code size in Darjeeling for some features we discuss below. This was determined by only counting the size of functions directly related to specific features. The actual cost is higher since some, especially garbage collection, also add complexity to other functions throughout the VM. Combined, the features below and the string functions mentioned in Section 1.1 make up about half the original Darjeeling VM.

Besides an increase in VM size, these features also cause a performance penalty, and features such as threads and exceptions are much harder in an AOT compiler where we cannot 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.

1.9.1 Threads

As shown in Table 1.3, support for threads accounts for about 10% of the VM size, if we exclude the string library. In addition, each thread requires a stack. 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 each stack as a linked list of frames on the heap. This is memory efficient, but allocating on the heap is slower and will occasionally trigger the

garbage collector.

A more cooperative concurrency model is more appropriate for sensor nodes, where lightweight tasks voluntarily yield the CPU and share a single stack. This is also the approach to concurrency chosen by a number of native code systems, including *t-kernel* [5], nesC [3], and more recently Amulet [8].

1.9.2 Exceptions

In terms of code size, exceptions are not very expensive to implement in an interpreter, but they are hard 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, since they could be easily replaced with return values to signal errors.

1.9.3 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 likely less than 2 KB, but the performance overhead is considerable. The target of a virtual method call must be resolved at run time, they cannot be made lightweight, and an AOT compiler can generate much more efficient code for calls to static methods.

In practice we seldom use 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 more lightweight form of function pointers that can be implemented efficiently. However, the details will require more careful study.

1.9.4 Garbage collection

Finally, garbage collection is clearly the most intrusive aspect of the JVM 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 1.3 shows the garbage

collector functions in Darjeeling add up to about 3.5 KB, 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 splits references and integers throughout the VM. This makes it easy for the garbage collector to find live references, but leads to significant code duplication and complexity. Using AOT compilation, the split stack adds overhead to maintain this state, and requires two extra registers as a second stack pointer that cannot be used for stack caching.

1.10 Building better sensor node VMs

In this chapter 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 presented here 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 has the advantage of being able to reuse existing knowledge and infrastructure.

However some of the issues 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, which would lead to more lightweight features and a more predictable memory model.

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

Chapter 2

Conclusion

This dissertation described the CapeVM sensor node virtual machine. CapeVM extends the state of the art by combining the desirable features of platform independent reprogramming, a safe execution environment, and acceptable performance. CapeVM was evaluated using a set of benchmarks, including small benchmarks to highlight specific behaviours, and five examples of real sensor node applications.

To come back to the research questions stated in Section **??**, we can conclude the following:

a. After identifying the sources of overhead in previous work on ahead-of-time compilers for sensor nodes, we introduced a number of optimisations to reduce the performance overhead by over 79%, and the code size overhead by 60%, resulting in an average performance overhead of 70%, and the code size overhead of 79% compared to native C.

   The optimisations introduced by CapeVM do increase the size of the VM, but the break-even point at which this is compensated for by the smaller code it generates, is well within the range of programme memory typically available on a sensor node.

   The price to pay for platform independence and a safe execution environment comes in three forms. There is still a performance overhead, but it is at a level that will be acceptable for many applications. The increase in code size however, and the space taken by the VM, do limit the size of applications we can load onto a device. Finally,

the overhead in memory usage is a problem for programmes allocating many small objects.

b. CapeVM's second contribution is providing a safe execution environment. Compared to native binary code, the higher level of abstraction of CapeVM's bytecode allowed us to develop a relatively simple set of safety checks.

   This results in a modest overhead in terms of VM size, and because most checks are performed at translation time, the overhead for providing safety is limited to a slowdown of 22% and a 2% increase in code size, compared to the unsafe version.

   Since to the best of our knowledge only two native code approaches exist that provide safety independent of the host, we cannot exclude the possibility that these could be further optimised. Currently however, CapeVM is on-par with or faster than existing systems, and provides platform independence at the same time.

c. Regarding the question of whether Java is a suitable language for a sensor node VM, we can conclude some aspects of it are a good match. An advantage of its simple stack-based instruction set is that it can be implemented in a small VM, and while we showed the stack-based architecture introduces significant overhead, most of this overhead is eliminated by our optimisations.

   However, our benchmarks also exposed several problems that ultimately make standard Java a poor choice. Specifically, the lack of support for constant data, and the inefficient use of memory for programmes containing many small objects meant some benchmarks could not be ported directly from C to Java. We proposed several improvements, and developed an extension to allow constant data to be put in flash memory, but conclude that more work is necessary to come to a more sensor node specific language and make sensor node VMs truly useful in a wide range of real-world projects.

Finally, we conclude by comparing our approach to existing work on improving sensor node VM performance, and on safe execution environments in Table 2.1.

Table 2.1: Comparison of our approach to related work

Approach	Platform indep.	Safe	Performance	Code size
Native code	No	No	1x	1x
Interpreters	Yes	Mostly no	300-23000% slower	50% smaller [b]
Ellul's AOT	Yes	No	123-811% slower	27-346% larger [b]
Safe TinyOS	No	Yes [a]	17% slower	27% larger
Harbor	No	Yes	160 to 1230% slower	30 to 65% larger
t-kernel	No	Yes	50 to 200% slower	500 to 750% larger
CapeVM (unsafe)	Yes	No	70% slower	79% larger
CapeVM (safe)	Yes	Yes	108% slower	83% larger

[a] requires a trusted host
[b] no support for constant data

Taking unsafe and platform specific native C as a baseline, we first note that existing interpreting sensor node VM's are typically not safe, and suffer from a 1 to 2 orders of magnitude slowdown. The performance overhead was reduced drastically by Ellul's work on Ahead-of-Time compilation, but still a significant overhead remains and this approach increases code size, reducing the size of programmes we can load onto a device.

On the safety side, Safe TinyOS achieves safety with relatively little overhead, but this depends on a trusted host. Harbor and *T-kernel* provide safety independent of the host, but at the cost of a significant performance overhead, or increase in code size respectively. Non of these approaches provide platform independence.

Finally, we see CapeVM provides both platform independence and safety, at a cost in terms of code size and performance that is lower than or comparable to previous work.

Appendix A

LEC benchmark source code

```
public class LEC {
    public static short benchmark_main(short[] numbers, Stream stream) {
        BSI bsi = new BSI();
        short ri_1 = 0;
        short NUMNUMBERS = (short)numbers.length;
        for (short i=0; i<NUMNUMBERS; i++) {
            short ri = numbers[i];
            compress(ri, ri_1, stream, bsi);

            ri_1 = ri;
        }

        // Return the number of bytes in the output array
        return (short)(stream.current_byte_index+1);
    }

    @ConstArray
    public static class si_tbl {
        public final static short data[] = {
            0b00, 0b010, 0b011, 0b100, 0b101, 0b110, 0b1110, 0b11110, 0b111110,
 ↪   0b1111110, 0b11111110, 0b111111110, 0b1111111110, 0b11111111110, 0b111111111110,
 ↪   0b1111111111110, 0b11111111111110
        };
    }

    @ConstArray
    public static class si_length_tbl {
        public static byte data[] = {
            2, 3, 3, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
        };
    }

    // pseudo code:
    // compress(ri, ri_1, stream)
    //     // compute difference di
    //     SET di TO ri - ri_1
    //     // encode difference di
    //     CALL encode() with di RETURNING bsi
    //     // append bsi to stream
    //     SET stream TO <<stream,bsi>>
    //     RETURN stream
    public static void compress(short ri, short ri_1, Stream stream, BSI bsi_obj) {
        // compute difference di
        short di = (short)(ri - ri_1);
        // encode difference di
        encode(di, bsi_obj);
        int bsi = bsi_obj.value;
        byte bsi_length = bsi_obj.length;
        // append bsi to stream
```

```
48          byte bits_left_current_in_byte = (byte)(8 - stream.bits_used_in_current_byte);
49          while (bsi_length > 0) {
50              if (bsi_length > bits_left_current_in_byte) {
51                  // Not enough space to store all bits
52
53                  // Calculate bits to write to current byte
54                  byte bits_to_add_to_current_byte =
55                          (byte)(bsi >> (bsi_length - bits_left_current_in_byte));
56
57                  // Add them to the current byte
58                  stream.data[stream.current_byte_index] |= bits_to_add_to_current_byte;
59                  // Remove those bits from the to-do list
60                  bsi_length -= bits_left_current_in_byte;
61
62                  // Advance the stream to the next byte
63                  stream.current_byte_index++;
64                  // Whole new byte for the next round
65                  bits_left_current_in_byte = 8;
66              } else {
67                  // Enough space to store all bits
68
69                  // After this we'll have -bsi_length bits left.
70                  bits_left_current_in_byte -= bsi_length;
71
72                  // Calculate bits to write to current byte
73                  byte bits_to_add_to_current_byte =
74                          (byte)(bsi << bits_left_current_in_byte);
75
76                  // Add them to the current byte
77                  stream.data[stream.current_byte_index] |= bits_to_add_to_current_byte;
78                  // Remove those bits from the to-do list
79                  bsi_length = 0;
80              }
81          }
82
83          stream.bits_used_in_current_byte = (byte)(8 - bits_left_current_in_byte);
84          // Note that if we filled the last byte, stream_bits_used_in_current_byte
85          // will be 8, which means in the next call to encode the first iteration of
86          // the while loop won't do anything, except advance the stream pointer.
87      }
88
89      // pseudo code:
90      // encode(di)
91      //     // compute di category
92      //     IF di = 0
93      //         SET ni to 0
94      //     ELSE
95      //         SET ni to CEIL(log_2(|di|))
96      //     ENDIF
97      //     // extract si from Table
98      //     SET si TO Table[ni]
99      //     // build bsi
100     //     IF ni = 0 THEN
101     //         // ai is not needed
102     //         SET bsi to si
103     //     ELSE
104     //         // build ai
105     //         IF di > 0 THEN
106     //             SET ai TO (di)|ni
107     //         ELSE
108     //             SET ai TO (di-1)|ni
109     //         ENDIF
110     //         // build bsi
111     //         SET bsi TO <<si,ai>>
112     //     ENDIF
113     //     RETURN bsi
114     private static void encode(short di, BSI bsi) {
115         // compute di category
116         short di_abs;
117         if (di < 0) {
118             di_abs = (short)-di;
119         } else {
120             di_abs = di;
```

```
        }
        byte ni = computeBinaryLog(di_abs);
        // extract si from Table
        short si = si_tbl.data[ni];
        byte si_length = si_length_tbl.data[ni];
        short ai = 0;
        byte ai_length = 0;
        // build bsi
        if (ni == 0) {
            bsi.value = si;
            bsi.length = si_length;
        } else {
            // build ai
            if (di > 0) {
                ai = di;
                ai_length = ni;
            } else {
                ai = (short)(di-1);
                ai_length = ni;
            }
            bsi.value = (si << ai_length) | (ai & ((1 << ni) -1));
            bsi.length = (byte)(si_length + ai_length);
        }
    }

    // pseudo code:
    // computeBinaryLog(di)
    //     // CEIL(log_r|di|)
    //     SET ni TO 0
    //     WHILE di > 0
    //         SET di TO di/2
    //         SET ni to ni + 1
    //     ENDWHILE
    //     RETURN ni
    private static byte computeBinaryLog(short di) {
        byte ni = 0;
        while (di > 0) {
            di >>= 1;
            ni++;
        }
        return ni;
    }
}

public class BSI {
    public int value;
    public byte length;
}

public class Stream {
    public Stream(short capacity) {
        data = new byte[capacity];
        current_byte_index = 0;
        bits_used_in_current_byte = 0;
    }

    public byte[] data;
    public short current_byte_index;
    public byte bits_used_in_current_byte;
}
```

Listing 3: LEC benchmark source code

Appendix B

Outlier detection benchmark source code

```
public class OutlierDetection {
    public static void benchmark_main(short NUMNUMBERS, short[] buffer, short[]
 ↪  distance_matrix, short distance_threshold, boolean[] outliers) {
        // Calculate distance matrix
        short sub_start=0;
        for (short i=0; i<NUMNUMBERS; i++) {
            short hor = sub_start;
            short ver = sub_start;
            for (short j=i; j<NUMNUMBERS; j++) {
                short buffer_i = buffer[i];
                short buffer_j = buffer[j];
                if (buffer_i > buffer_j) {
                    short diff = (short)(buffer_i - buffer_j);
                    distance_matrix[hor] = diff;
                    distance_matrix[ver] = diff;
                } else {
                    short diff = (short)(buffer_j - buffer_i);
                    distance_matrix[hor] = diff;
                    distance_matrix[ver] = diff;
                }

                hor ++;
                ver += NUMNUMBERS;
            }
            sub_start+=NUMNUMBERS+1;
        }

        // Determine outliers
        // Since we scan one line at a time, we don't need to calculate
        // a matrix index. The first NUMNUMBERS distances correspond to
        // measurement 1, the second NUMNUMBERS distances to measurement 2, etc.
        short k=0;
        short half_NUMNUMBERS = (short)(NUMNUMBERS >> 1);
        // This is necessary because Java doesn't have unsigned types
        if (distance_threshold > 0) {
            for (short i=0; i<NUMNUMBERS; i++) {
                short exceed_threshold_count = 0;
                for (short j=0; j<NUMNUMBERS; j++) {
                    short diff = distance_matrix[k++];
                    if (diff < 0 || diff > distance_threshold) {
                        exceed_threshold_count++;
                    }
                }

```

```
                if (exceed_threshold_count > half_NUMNUMBERS) {
                    outliers[i] = true;
                } else {
                    outliers[i] = false;
                }
            }
        } else {
            for (short i=0; i<NUMNUMBERS; i++) {
                short exceed_threshold_count = 0;
                for (short j=0; j<NUMNUMBERS; j++) {
                    short diff = distance_matrix[k++];
                    if (diff < 0 && diff > distance_threshold) {
                        exceed_threshold_count++;
                    }
                }

                if (exceed_threshold_count > half_NUMNUMBERS) {
                    outliers[i] = true;
                } else {
                    outliers[i] = false;
                }
            }
        }
    }
}
```

Listing 4: Outlier detection benchmark source code

Appendix C

Heat detection benchmark source code

C.1 Calibration

```
public class HeatCalib {
    public static short[] ACal;
    public static int[] QCal;
    public static short[] stdCal;
    public static short[] zscore;
    public static short z_min, z_max;

    @Lightweight
    public static native void get_heat_sensor_data(short[] frame_buffer,
                                                   short frame_number);

    public static void benchmark_main() {
        short[] frame_buffer = new short[64];

        for (short i=0; i<100; i++) {
            get_heat_sensor_data(frame_buffer, i);
            fast_calibration(frame_buffer, i);
        }
        get_heat_sensor_data(frame_buffer, (short)100);
        zscoreCalculation(frame_buffer);
    }

    private static void fast_calibration(short[] frame_buffer, short frame_number) {
        short frame_number_plus_one = (short)(frame_number+1);
        for(short i=0; i<64; i++) {
            short previous_ACal = ACal[i];
            ACal[i] += (frame_buffer[i] - ACal[i] + (frame_number_plus_one >>> 1)
                        ) / frame_number_plus_one;
            QCal[i] += (frame_buffer[i] - previous_ACal) * (frame_buffer[i] - ACal[i]);
        }
        for(short i=0; i<64; i++) {
            stdCal[i] = isqrt(QCal[i]/frame_number_plus_one);
        }
    }

    // http://www.cc.utah.edu/~nahaj/factoring/isqrt.c.html
    @Lightweight
    private static short isqrt (int x) {
        int   squaredbit, remainder, root;

        if (x<1) return 0;

        /* Load the binary constant 01 00 00 ... 00, where the number
        * of zero bits to the right of the single one bit
```

```
        * is even, and the one bit is as far left as is consistant
        * with that condition.)
        */
        squaredbit  = (int) ((((int) ~0L) >>> 1) &
                       ~(((int) ~0L) >>> 2));
        /* This portable load replaces the loop that used to be
        * here, and was donated by  legalize@xmission.com
        */

        /* Form bits of the answer. */
        remainder = x;  root = 0;
        while (squaredbit > 0) {
            if (remainder >= (squaredbit | root)) {
                remainder -= (squaredbit | root);
                root >>= 1; root |= squaredbit;
            } else {
                root >>= 1;
            }
            squaredbit >>= 2;
        }

        return (short)root;
    }

    private static void zscoreCalculation(short[] frame_buffer) {
        short tempMax = -30000;
        short tempMin = 30000;

        for(int i=0; i<64; i++) {
            short score = (short)(100 * (frame_buffer[i] - ACal[i]) / stdCal[i]);

            zscore[i] = score;

            if(score > tempMax) {
                tempMax = score;
            }

            if(score < tempMin) {
                tempMin = score;
            }
        }

        z_max = tempMax;
        z_min = tempMin;
    }
}
```

Listing 5: Heat detection benchmark source code (calibration phase)

C.2 Detection

```
package javax.rtcbench;

import javax.rtc.RTC;
import javax.rtc.Lightweight;

public class HeatDetect {
    public static final byte THRESHOLD_LEVEL1 = 2;
    public static final byte THRESHOLD_LEVEL2 = 3;
    public static final byte RED              = 4;
    public static final byte ORANGE           = 3;
    public static final byte YELLOW           = 2;
    public static final byte WHITE            = 1;

    public static final byte ARRAY_SIZE       = 8;
    public static final byte WAITTOCHECK      = 90;
    public static final byte CHECKED          = 91;
```

```
    public static final byte NEIGHBOR         = 92;
    public static final byte BOUNDARY         = 99;

    public static final byte LEFT_UP          = 7;
    public static final byte RIGHT_UP         = 63;
    public static final byte LEFT_DOWN        = 0;
    public static final byte RIGHT_DOWN       = 56;

    public static int x_weight_coordinate = 0;
    public static int y_weight_coordinate = 0;
    public static int xh_weight_coordinate = 0;
    public static int yh_weight_coordinate = 0;

    public static int yellowGroupH = 0;
    public static int yellowGroupL = 0;
    public static int orangeGroupH = 0;
    public static int orangeGroupL = 0;
    public static int redGroupH = 0;
    public static int redGroupL = 0;

    public static boolean[] zscoreWeight = null;
    private static short[] neighbor = new short[8];

    private static final short zscore_threshold_high = 1000;
    private static final short zscore_threshold_low = 500;
    private static final short zscore_threshold_hot = 5000;
    private static final short zscore_threshold_recal = 1000;


    public static void benchmark_main(short[] frame_buffer,
                                      byte[] color,
                                      byte[] rColor,
                                      int[] largestSubset,
                                      int[] testset,
                                      int[] result) {
        HeatCalib.zscoreCalculation(frame_buffer);

        ShortWrapper maxSubsetLen = new ShortWrapper();
        maxSubsetLen.value = 0;
        get_largest_subset(largestSubset, maxSubsetLen, testset, result);

        reset_log_variable(color);

        if (maxSubsetLen.value > 1) {
            if (HeatCalib.z_max > zscore_threshold_hot) {
                get_filtered_xy(largestSubset, maxSubsetLen.value);
            } else if (HeatCalib.z_max > zscore_threshold_low) {
                get_xy(largestSubset, maxSubsetLen.value);
            }
            labelPixel(largestSubset, maxSubsetLen.value, color);
            rotateColor(color, rColor);
            findGroup(rColor);
        } else {
            RTC.avroraBreak();
        }
    }

    private static void get_largest_subset(int[] largestSubset,
                                           ShortWrapper maxSubsetLen,
                                           int[] testset,
                                           int[] result) {
        int pixelCount=0;
        for(short i=0; i<64; i++){
            testset[i]=0;
        }
        for(short i=0; i<64; i++){
            if (HeatCalib.zscore[i] > zscore_threshold_high) {
                testset[pixelCount]=i;
                pixelCount++;
                zscoreWeight[i]=true;
            } else if (HeatCalib.zscore[i] > zscore_threshold_low) {
                if (zscoreWeight[i] || check_neighbor_zscore_weight(i)) {
                    testset[pixelCount]=i;
```

```
                    pixelCount++;
                    zscoreWeight[i]=true;
                }
            } else {
                zscoreWeight[i]=false;
            }
        }
        testset[pixelCount] = BOUNDARY;

        find_largestSubset(testset, pixelCount, maxSubsetLen, largestSubset, result);

        // If subset only has one pixel higher than zscore threshold,
        // it will be considereddd as a noise.
        // This subset will be reset here.
        if (maxSubsetLen.value == 1) {
          maxSubsetLen.value = 0;
          largestSubset[0] = -1; // reset
        }
    }

    private static void find_largestSubset(int[] testset,
                                           int testsetLen,
                                           ShortWrapper maxSubsetLen,
                                           int[] largestSubset,
                                           int[] result){
        for(short i=0; i<64;i++){result[i]=WAITTOCHECK;}
        int subsetNumber = 0;
        ShortWrapper startIndex = new ShortWrapper();
        startIndex.value = 0;
        while(get_startIndex(testset, testsetLen, result, startIndex)){
            result[testset[startIndex.value]]=subsetNumber;
            label_subset(testset, testsetLen, result, subsetNumber);
            subsetNumber++;
        }
        select_largest_subset(testset, testsetLen, result,
                              subsetNumber, maxSubsetLen, largestSubset);
    }

    private static void select_largest_subset(int[] testset,
                                              int testsetLen,
                                              int[] result,
                                              int subsetNumber,
                                              ShortWrapper maxSubsetLen,
                                              int[] largestSubset){
        int maxSubsetNumber = 0;
        for(short i=0; i<subsetNumber; i++){
            short lengthCount = 0;
            for(short j=0; j<testsetLen; j++){
                if(result[testset[j]] == i){
                    lengthCount++;
                }
            }
            if(lengthCount > maxSubsetLen.value){ // if equal, no solution currently
                maxSubsetLen.value = lengthCount;
                maxSubsetNumber = i;
            }
        }

        // largestSubset = (int*)malloc(sizeof(int)*(*maxSubsetLen));
        short index=0;
        for(short i=0; i<64; i++){
            if(result[i]==maxSubsetNumber){
                largestSubset[index]=i;
                index++;
            }
        }
    }


    private static void reset_log_variable(byte[] color)
    {
        for(short i=0; i<64; i++) {
            color[i] = WHITE;
```

```
        }

        x_weight_coordinate = -1;
        y_weight_coordinate = -1;
        xh_weight_coordinate = -1;
        yh_weight_coordinate = -1;

        yellowGroupH = 0;
        yellowGroupL = 0;
        orangeGroupH = 0;
        orangeGroupL = 0;
        redGroupH = 0;
        redGroupL = 0;
    }

    private static void get_filtered_xy(int[] largestSubset, int maxSubsetLen) {
        short x_weight_zscore_sum = 0;
        short y_weight_zscore_sum = 0;
        int zscore_sum = 0;
        byte low_zscore_length = 0;

        short xh_weight_zscore_sum = 0;
        short yh_weight_zscore_sum = 0;
        int zscore_sum_h = 0;
        byte hot_zscore_length = 0;

        for(short i=0; i<maxSubsetLen; i++) {
            short _zscore = HeatCalib.zscore[largestSubset[i]];
            if (_zscore > zscore_threshold_hot) {
                zscore_sum_h += _zscore;
                hot_zscore_length += 1;
            } else if (_zscore > zscore_threshold_low) {
                zscore_sum += _zscore;
                low_zscore_length += 1;
            }
        }

        for(short i=0; i<maxSubsetLen; i++) {
            int _x = largestSubset[i] % 8;
            int _y = largestSubset[i] >>> 3;
            short _zscore = HeatCalib.zscore[largestSubset[i]];
            if (_zscore > zscore_threshold_hot) {
                xh_weight_zscore_sum += _x * _zscore / zscore_sum_h;
                yh_weight_zscore_sum += _y * _zscore / zscore_sum_h;
            } else if (_zscore > zscore_threshold_low) {
                x_weight_zscore_sum += _x * _zscore / zscore_sum;
                y_weight_zscore_sum += _y * _zscore / zscore_sum;
            }
        }

        if (hot_zscore_length > 0) {
            xh_weight_coordinate = xh_weight_zscore_sum;
            yh_weight_coordinate = yh_weight_zscore_sum;
        }

        if (low_zscore_length > 0) {
            x_weight_coordinate = x_weight_zscore_sum;
            y_weight_coordinate = y_weight_zscore_sum;
        }
    }

    private static void get_xy(int[] largestSubset, int maxSubsetLen) {
        short x_weight_zscore_sum = 0;
        short y_weight_zscore_sum = 0;
        int zscore_sum = 0;

        for(short i=0; i<maxSubsetLen; i++) {
            short _zscore = HeatCalib.zscore[largestSubset[i]];
            zscore_sum += _zscore;
        }

        for(short i=0; i<maxSubsetLen; i++) {
            int _x = largestSubset[i] % 8;
```

```
            int _y = largestSubset[i] >>> 3;
            short _zscore = HeatCalib.zscore[largestSubset[i]];
            x_weight_zscore_sum += 100 * _x * _zscore / zscore_sum;
            y_weight_zscore_sum += 100 * _y * _zscore / zscore_sum;
        }
        x_weight_coordinate = x_weight_zscore_sum;
        y_weight_coordinate = y_weight_zscore_sum;
    }


    private static void labelPixel(int[] largestSubset, int maxSubsetLen, byte[] color)
    {
        for(short i=0; i<maxSubsetLen; i++) {
            int pixelIndex = largestSubset[i];
            if (HeatCalib.zscore[pixelIndex] > zscore_threshold_hot) {
                color[pixelIndex] = RED;
            } else if (HeatCalib.zscore[pixelIndex] > zscore_threshold_high) {
                color[pixelIndex] = ORANGE;
            } else if (HeatCalib.zscore[pixelIndex] > zscore_threshold_low) {
                color[pixelIndex] = YELLOW;
            }
        }
    }

    private static void rotateColor(byte[] color, byte[] rColor)
    {
        for(short i=0; i<8; i++) {
            for (short j=0; j<8; j++) {
                rColor[(i<<3)+j] = color[LEFT_UP + (j<<3) - i];
            }
        }
    }

    private static void findGroup(byte[] color)
    {
        for(short i=0; i<32; i++){
            byte cl = color[i];
            if(cl == YELLOW){
                yellowGroupL |= 1<<i;
            }else if(cl == ORANGE){
                orangeGroupL |= 1<<i;
            }else if(cl == RED){
                redGroupL |= 1<<i;
            }
            cl = color[i+32];
            if(cl == YELLOW){
                yellowGroupH |= 1<<i;
            }else if(cl == ORANGE){
                orangeGroupH |= 1<<i;
            }else if(cl == RED){
                redGroupH |= 1<<i;
            }
        }
    }

    private static boolean get_startIndex(int[] testset,
                                          int testsetLen,
                                          int[] result,
                                          ShortWrapper startIndex) {
        boolean rv = false; // done this way to avoid values on the stack at a brtarget
        for(short i=0; i<testsetLen; i++){
            if(result[testset[i]] == WAITTOCHECK){
                startIndex.value = i;
                rv = true;
                break;
            }
        }
        return rv;
    }

    private static void label_subset(int[] testset,
                                     int testsetLen,
                                     int[] result,
```

```
                                        int subsetNumber) {
        while(label_neighbor(result, subsetNumber)){
            for(short i=0; i< testsetLen; i++){
                if(result[testset[i]] == NEIGHBOR){
                    result[testset[i]]=subsetNumber;
                }
            }
            for(short i=0; i<64; i++){
                if(result[i] == NEIGHBOR){
                    result[i]=CHECKED;
                }
            }
        }
    }

    private static void get_eight_neighbor(short loc, short[] neighbor) //neighbor length maximum is 8
    {
        for (short i=0; i<8; i++) {
            neighbor[i] = -1;
        }
        if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==0){
            neighbor[0]=(short)(loc+1); neighbor[1]=(short)(loc+ARRAY_SIZE); neighbor[2]=(short)(loc+ARRAY_SIZE+1);
        }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)>0 && (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+1); neighbor[2]=(short)(loc+ARRAY_SIZE-1); neighbor[3]=(short)(loc+ARRAY_SIZE); neighbor[4]=(short)(loc+ARRAY_SIZE+1);
        }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+ARRAY_SIZE-1); neighbor[2]=(short)(loc+ARRAY_SIZE);
        }else if((loc >>> 3)> 0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
            neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-ARRAY_SIZE+1); neighbor[2]=(short)(loc+1); neighbor[3]=(short)(loc+ARRAY_SIZE); neighbor[4]=(short)(loc+ARRAY_SIZE+1);
        }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 && (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-ARRAY_SIZE-1); neighbor[1]=(short)(loc-ARRAY_SIZE); neighbor[2]=(short)(loc-ARRAY_SIZE+1); neighbor[3]=(short)(loc-1); neighbor[4]=(short)(loc+1); neighbor[5]=(short)(loc+ARRAY_SIZE-1); neighbor[6]=(short)(loc+ARRAY_SIZE); neighbor[7]=(short)(loc+ARRAY_SIZE+1);
        }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-ARRAY_SIZE-1); neighbor[1]=(short)(loc-ARRAY_SIZE); neighbor[2]=(short)(loc-1); neighbor[3]=(short)(loc+ARRAY_SIZE-1); neighbor[4]=(short)(loc+ARRAY_SIZE);
        }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
            neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-ARRAY_SIZE+1); neighbor[2]=(short)(loc+1);
        }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 && (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-ARRAY_SIZE-1); neighbor[1]=(short)(loc-ARRAY_SIZE); neighbor[2]=(short)(loc-ARRAY_SIZE+1); neighbor[3]=(short)(loc-1); neighbor[4]=(short)(loc+1);
        }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-ARRAY_SIZE-1); neighbor[1]=(short)(loc-ARRAY_SIZE); neighbor[2]=(short)(loc-1);
        }
    }

    private static void get_four_neighbor(short loc, short[] neighbor) //neighbor length maximum is 4
    {
        for (short i=0; i<4; i++) {
            neighbor[i] = -1;
        }
        if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==0){
            neighbor[0]=(short)(loc+1); neighbor[1]=(short)(loc+ARRAY_SIZE);
        }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)>0 && (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
            neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+1); neighbor[2]=(short)(loc+ARRAY_SIZE);
```

```
359             }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
360                 neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+ARRAY_SIZE);
361             }else if((loc >>> 3)> 0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
362                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc+1);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE);
363             }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 &&
    ↪   (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
364                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
    ↪   neighbor[2]=(short)(loc+1); neighbor[3]=(short)(loc+ARRAY_SIZE);
365             }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc %
    ↪   ARRAY_SIZE)==(ARRAY_SIZE-1)){
366                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE);
367             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
368                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc+1);
369             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 && (loc %
    ↪   ARRAY_SIZE)<(ARRAY_SIZE-1)){
370                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
    ↪   neighbor[2]=(short)(loc+1);
371             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
372                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
373             }
374         }
375 
376         private static boolean label_neighbor(int[] result, int subsetNumber){
377             boolean hasNeighbor = false;
378             for(short i=0; i<64; i++){
379                 if(result[i]==subsetNumber){
380                     get_eight_neighbor(i, neighbor);
381                     for(short j=0; j<8;j++){
382                         if(neighbor[j] != -1 && result[neighbor[j]] == WAITTOCHECK){
383                             result[neighbor[j]]=NEIGHBOR;
384                             hasNeighbor = true;
385                         }
386                     }
387                 }
388             }
389             return hasNeighbor;
390         }
391 
392         private static boolean check_neighbor_zscore_weight(short index) {
393             boolean rv = false;  // done this way to avoid values on the stack at a brtarget
394             get_four_neighbor(index, neighbor);
395             for(short i=0; i<4;i++){
396                 if (neighbor[i] != -1) {
397                     if (zscoreWeight[neighbor[i]]) {
398                         rv = true;
399                         break;
400                     }
401                 }
402             }
403             return rv;
404         }
405     }
406 
407     public class ShortWrapper {
408         public short value;
409     }
```

Listing 6: Heat detection benchmark source code (detection phase)

Bibliography

[1] F. Aslam. *Challenges and Solutions in the Design of a Java Virtual Machine for Resource Constrained Microcontrollers*. PhD thesis, University of Freiburg, 2011.

[2] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. *OOPSLA '99: Proceedings of the Fourteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov. 1999.

[3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language:A Holistic Approach to Networked Embedded Systems. *PLDI '03: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[4] B. Goetz. Java theory and practice: Urban performance legends, revisited. *IBM developerWorks*, Sept. 2005. `https://www.ibm.com/developerworks/library/j-jtp09275/j-jtp09275-pdf.pdf`.

[5] L. Gu and J. A. Stankovic. t-kernel: A Translative OS Kernel for Wireless Sensor Networks. Technical Report UVA CS TR CS-2005-09, University of Virginia, June 2005.

[6] Guard Square. ProGuard 5.3.2. `https://sourceforge.net/projects/proguard`, Dec. 2016.

[7] T. Harbaum. NanoVM. `http://harbaum.org/till/nanovm/index.shtml`, June 2006.

[8] J. Hester, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, K. Freeman, S. Lord, R. Halter, D. Kotz, and J. Sorber. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. *SenSys '16: Proceedings of the Fourteenth International Conference on Embedded Networked Sensor Systems*, Nov. 2016.

[9] Oracle. Connected Limited Device Configuration (CLDC); JSR 139. `http://www.oracle.com/technetwork/java/cldc-141990.html`, Feb. 2005.